Progress in Estimation and Control for Air-Launched Missiles:

Part I: A Real-Time 3D Mini-Max Pursuit-Evader Algorithm
Part II: Estimation

Final Technical Report
For the Period: 15 August 1992 through 31 October 1995
Contract Number: F49620-92-C-0056

AFOSR-TR-96

0184

Prepared for:

Air Force Office of Scientific Research

Bolling Air Force Base, DC 20332

Prepared by:

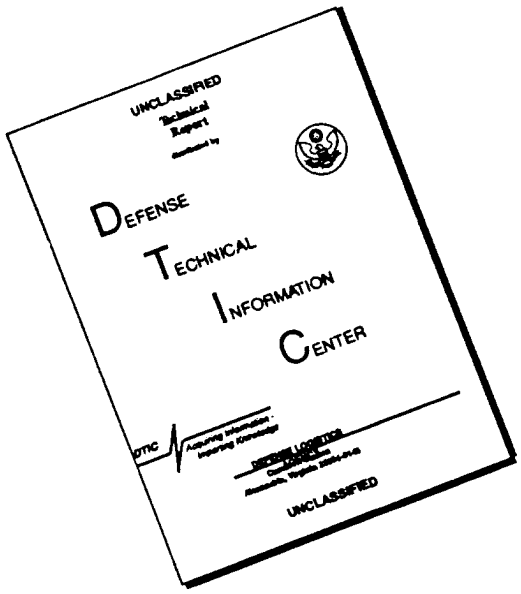
Honeywell Technology Center

3660 Technology Drive

Minneapolis, MN 55418

December 1995

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	12/27/95	Final Report, 8/92 to 10/95

4. TITLE AND SUBTITLE

Title: Progress in Estimation and Control for Air-Launched Missiles
Subtitle: I: A Real-Time 3D Mini-Max Pursuit-Evader Algorithm, II: Estimation

5. FUNDING NUMBERS

6. AUTHOR(S)

Mike Elgersma and Blaise Morton

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Honeywell Technology Center
Honeywell Inc.
3660 Technology Drive
Minneapolis, Minnesota 55418

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Office of Scientific Research
Bolling Air Force Base, DC 20332

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTAL NOTES

12A. DISTRIBUTION/AVAILABILITY STATEMENT

Unlimited

12b. DISTRIBUTION CODE

Unlimited

13. ABSTRACT (Maximum 200 words)

Part I of this report describes a real-time 3D mini-max pursuit-evader algorithm. The fast algorithm is possible due to the assumption of constant speed, piecewise circular trajectories.

Part II of this report includes several papers on the interaction of estimation and control for missile guidance.

14. SUBJECT TERMS

Real-Time, 3D, Mini-Max, Pursuit-Evader, Estimation

15. NUMBER OF PAGES

120

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

Preface

I would like to thank Jim Krause and Tom Ting from Honeywell for initiating this contract and working the first two years of it. Progress for those first two years is documented in part II of this final report.

I would like to thank Marc Jacobs at AFOSR for funding this three-year study on control and estimation of air-launched missiles. I would also like to thank the Initiatives Committee at Honeywell for funding a related study of missiles pursuing highly maneuverable aircraft.

Michael R. Elgersma

December 28, 1995

TABLE OF CONTENTS

1. Introduction

Many studies have been done on various forms of mini-max pursuit-evader games for missiles and aircraft. These types of algorithms typically give performance that is superior to simpler missile guidance algorithms such as proportional navigation or linear control techniques. The bibleography gives titles and abstracts of nine papers about mini-max pursuit-evader algorithms, as well as one paper on a nonlinear regulator algorithm that could be useful for missile guidance.

The 3D mini-max pursuit-evader algorithms typically run far too slow for real-time implementations, so they are restricted to simulation use. Since they are run off-line, they can include highly detailed dynamics and constraints, at the expense of run-time.

In this report, we derive a 3D mini-max pursuit-evader algorithm that takes between 250,000 and 2 million floating point operations to update. This guidance algorithm could be updated at around 1 Hz on a rather slow 1 Mflop computer. In order to get this performance, we had to assume that both vehicles moved at (different) piece-wise constant speeds, along piece-wise circular 3D trajectories. This leaves four piece-wise constant control inputs: the two normal accelerations of each of the vehicles. Conventional mini-max algorithms would have to search through a four-dimensional control space to find the optimal trajectories.

With the assumptions of constant speeds and constant accelerations, we have derived the implicit equations determining the two-dimensional surface in 3D on which all possible intercepts occur. The two-dimensional intercept surface typically encloses the slower aircraft, allowing it no escape. Fuel and acceleration constraints on the missile leave openings in the two-dimensional intercept surface through which the evading aircraft can escape. Our guidance algorithm gets its speed by using either a closed-form solution (2 million floating point operations) or an iterative solution (250,000 floating point operations) for all points on a

30×60 grid on the two-dimensional surface of all possible intercept points in 3D space.

At each update (e.g. 2 Hz), the algorithm inputs the current velocity of both vehicles and the relative position between them. From this information, the missile computes what acceleration input it needs to minimize the miss distance for each possible aircraft acceleration input. This results in a 2-dimensional intercept surface, parameterized by the aircraft's 2-dimensional acceleration input. On this surface, the aircraft computes some cost functional (e.g. some combination of intercept time, max missile acceleration, etc.) that it wants to maximize and choses the corresponding aircraft acceleration input. Then the missile computes the acceleration required for the missile to get to that point on the intercept surface.

Given the circular trajectory from the missile's initial position to the desired end-point, with the missile's initial velocity tangent to the circle, there is no other trajectory that simultaneously has less maximum-acceleration and less flight time. We ignore paths of longer length which can have smaller curvature. By using the smallest reasonable intercept acceleration (circular paths), the missile avoids its own acceleration limits, and is less prone to hit its angular rate limits chasing a wildly maneuvering aircraft.

A disadvantage of the proposed algorithm (compared to proportional nav) is that it requires measurements of the velocity of both vehicles, and range. Since this information is seldom available from sensors on a small missile, the missile may have to rely on getting these information updates from the aircraft that launched the missile. Alternatively, the algorithm could be used only to display the intercept surface to the pursuing aircraft to let the pilot know when to launch a missile. Another alternative is to use the algorithm on the evading aircraft to let it know what point on (or hole in) the intercept surface it should head toward to best avoid a missile that has been launched at it.

Included near the end of this report are some examples that were run on a 120 MHz Pentium.

The guidance algorithm itself ran at approximately 80 Hz. The color figures (see figures 10 through 18) are snapshots of the moving graphics done using OpenGL which Microsoft has licensed from Silicon Graphics for the Windows NT operating system.

2. Equations of Motion and Mini-Max Formulation

Let vehicle 1 be the evader, and vehicle 2 be the pursuer.

Let $\underline{r}_i(t) \in \mathbf{R}^3$ be the position of vehicle i.

Let $\underline{V}_i(t) \in \mathbf{R}^3$ be the velocity of vehicle i.

Let m_i be the mass of vehicle i.

Let $\underline{u}_i \in \mathbf{U}_i \subset \mathbf{R}^2$ be the two perpendicular forces on vehicle i.

Let $\underline{V}_i^{\perp}(t)$ be the orthonormal (orthogonal) complement of $\underline{V}_i(t)$.

The constant speed, point-mass equations of motion, in inertial coordinates, are given by:

$$\begin{aligned} m_i \dot{\underline{V}}_i(t) &= \underline{V}_i^{\perp}(t)\, \underline{u}_i \\ \dot{\underline{r}}_i(t) &= \underline{V}_i(t) \qquad\qquad i=1,2 \end{aligned} \tag{1}$$

with initial conditions:

$$\underline{V}_i(0) \qquad \underline{r}_i(0) \qquad i=1,2 \tag{2}$$

We are considering mini-max formulations of the following type for determining the optimal control inputs $\underline{u}_i = \underline{u}_i^*$. The pursuer tries to minimize the miss distance, while the evader tries to minimize some cost functional J.

$$[T^*(\underline{u}_1),\ \underline{u}_2^*(\underline{u}_1)] = \text{min\_args}\left[\min_{\substack{0\le t<T_{max} \\ \underline{u}_2 \in \mathbf{U}_2 \subset \mathbf{R}^2}} \|\underline{r}_2(t) - \underline{r}_1(t)\| \right] \qquad \text{miss distance} \tag{3}$$

$$\underline{u}_1^* = \arg\left[\max_{\underline{u}_1 \in \mathbf{U}_1 \subset \mathbf{R}^2} \left[J\left[T^*(\underline{u}_1),\ \underline{u}_2^*(\underline{u}_1),\ \underline{r}_1(T^*(\underline{u}_1)),\ \underline{r}_2(T^*(\underline{u}_1)) \right] \right] \right] \qquad \text{cost index} \tag{4}$$

$$\underline{u}_2^* = \underline{u}_2^*(\underline{u}_1^*) \tag{5}$$

The mini-max computation is updated as fast as the guidance computer can calculate it, using the current state, $\underline{r}_i(t)$, $\underline{V}_i(t)$ as the new initial conditions.

If the mini-max algorithm is only evaluated once, both vehicles will have constant control inputs, and will fly along circular trajectories. Each time the mini-max algorithm is updated, each vehicle recomputes the optimal value of its control inputs. This results in each vehicle's trajectory consisting of smoothly connected circular segments.

For some initial conditions $\underline{r}_i(0)$, $\underline{V}_i(0)$, large enough values of T_{max}, and large enough control-input sets $\mathbf{U}_i$, the vehicles intercept (zero miss distance) for any value of $\underline{u}_1 \in \mathbf{U}_1 \subset \mathbf{R}^2$, i.e.:

$$\left[\min_{\substack{0 \le t < T_{max} \\ \underline{u}_2 \in \mathbf{U}_2 \subset \mathbf{R}^2}} \| \underline{r}_2(t) - \underline{r}_1(t) \| \right] = 0 \tag{6}$$

In this case, $T^*(\underline{u}_1)$ is the intercept time, and $\underline{r}_1(T^*(\underline{u}_1)) = \underline{r}_2(T^*(\underline{u}_1))$ is the intercept point. If more than one intercept occurs, $T^*(\underline{u}_1)$ is the smallest of those intercept times. The function $\underline{r}_1(T^*(\underline{u}_1))$ maps the 2-dimensional control space $\mathbf{U}_1$ into $\mathbf{R}^3$, giving the 2-dimensional intercept surface in 3-dimensional space.

Examples of the evader's cost functional J could be:

$J = T^*(\underline{u}_1)$ maximize intercept time, depleting pursuer's fuel

$J = \| \underline{u}_2^*(\underline{u}_1) \|$ maximize pursuer acceleration, putting pursuer on acceleration limits (7)

$J = \| \underline{r}_2(T^*(\underline{u}_1)) - \underline{r}_1(T^*(\underline{u}_1)) \|$ maximize miss distance (if non–zero)

Circular Paths have Minimum Max-Acceleration

Given the unique circular trajectory from the missile's initial position to the desired end-point, with the missile's initial velocity tangent to the circle, there is no other trajectory that simultaneously has less maximum-acceleration and less flight time. We ignore paths of longer length which can have smaller curvature. By using the smallest reasonable intercept acceleration (circular paths), the missile avoids its own acceleration limits, and is less prone to hit its angular rate limits chasing a wildly maneuvering aircraft.

For a proof that min-length paths with a local curvature bound consist of a circular segment and a straight segment, see [Dubins] whose work was also sponsored by AFOSR.

3. Intercept Surface for Vehicles on Constant-Speed 3D Circular Trajectories

3.1 Introduction

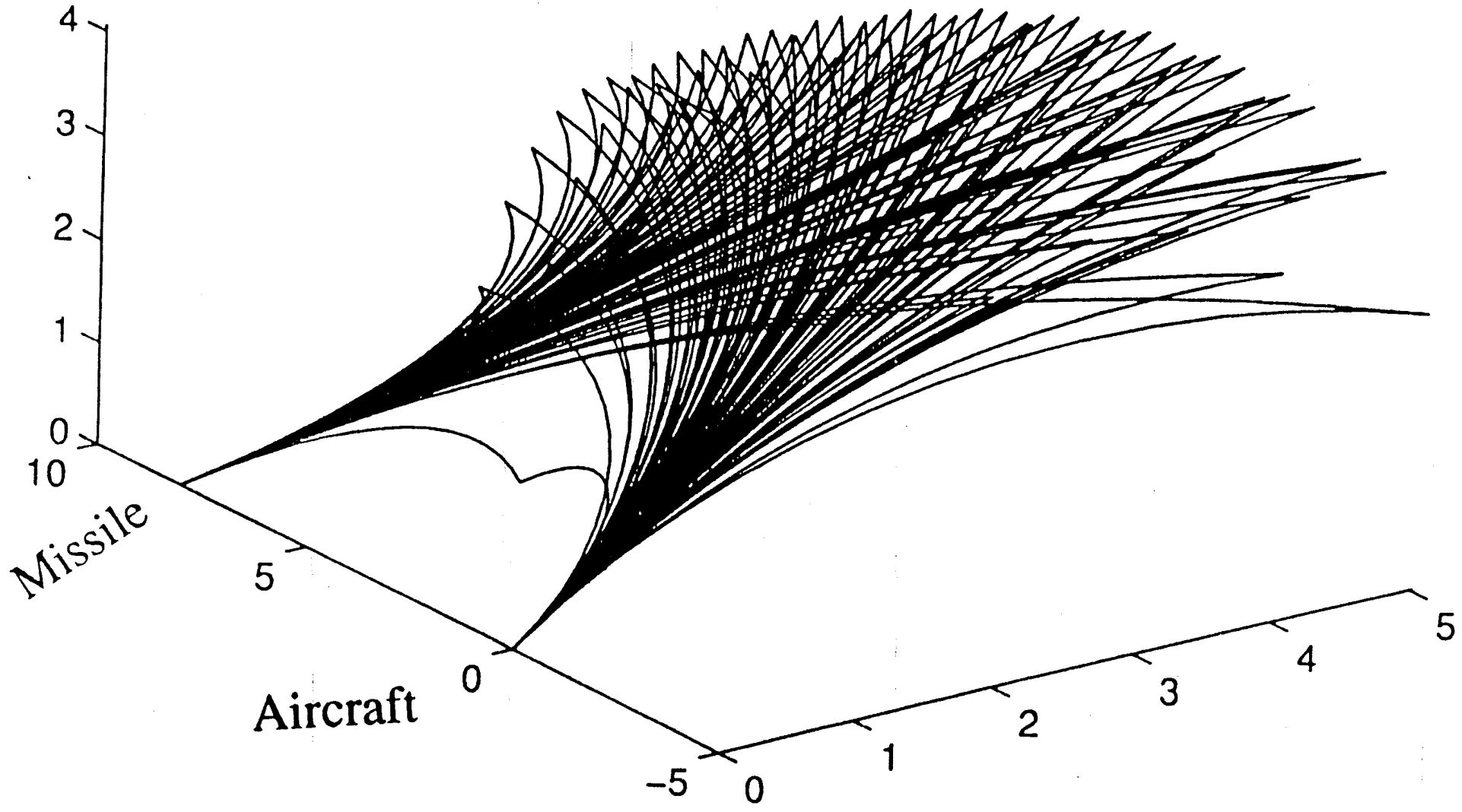
If speed and acceleration are allowed to vary within some bounded region, the associated mini-max evader-pursuer problem is computationally expensive and unlikely to be solved in real-time by today's computers. By assuming constant speed and any allowed constant normal acceleration, the problem becomes tractable. In later sections, we will partially alleviate the restrictions of constant speed and constant normal acceleration, by recomputing the current optimum "constant" accelerations using the current positions and speeds, as the trajectories evolve.

In this section, we will derive the equations that define the intercept surface for two vehicles on constant-speed circular trajectories. In the next two sections we will give algorithms for solving the resulting set of implicit equations.

Assume the evader (aircraft) has initial location $\underline{r}_1(0)$ and initial velocity $\underline{V}_1(0)$. Assume the pursuer (missile) has initial location $\underline{r}_2(0)$ and initial velocity $\underline{V}_2(0)$.

If both vehicles maintain constant speed and constant normal acceleration, they both will remain on circular paths in 3D space. The set of constant-speed circular trajectories through the point $\underline{r}_1(0)$, with initial velocity $\underline{V}_1(0)$ is 2-dimensional, and could be parameterized by the two components of normal acceleration. Similarly, the set of constant-speed circular trajectories through the point $\underline{r}_2(0)$, with initial velocity $\underline{V}_2(0)$ is 2-dimensional, and could be parameterized by the two components of normal acceleration. The constraint that the circles for vehicle$_1$ and vehicle$_2$ intersect reduces the 4-dimensional set to a 3-dimensional set. The constraint that vehicle$_1$ and vehicle$_2$ arrive at the intersection point at the same time, reduces the intercept surface to a 2-dimensional set. Figure 1 shows typical intercepts.

Figure 1: Intercepts Along Circular Paths



This two-dimensional surface separates 3D space into regions that the aircraft can get to first, and regions that the missile can get to first. If the missile is capable of catching the aircraft, then the regions that the aircraft can get to first include a bounded region in front of the aircraft that shrinks to a point as the missile approaches intercept.

To solve for the locations $\underline{r}$ in $\mathbf{R}^3$ where the vehicles can intercept, we will compute the time it takes vehicle$_1$ and vehicle$_2$ to get to point $\underline{r}$ via circular paths from $\underline{r}_1(0)$ and $\underline{r}_2(0)$.

We will define two sets of polar coordinates, based at the points $\underline{r}_1(0)$ and $\underline{r}_2(0)$.

$$\begin{aligned} \underline{r} &= \underline{r}_1(0) + r_1 \left[\frac{\underline{V}_1(0)}{\|\underline{V}_1(0)\|}, \underline{V}_1(0)^{\perp} \right] \begin{bmatrix} \cos(\eta_1) \\ \sin(\eta_1)\cos(\zeta_1) \\ \sin(\eta_1)\sin(\zeta_1) \end{bmatrix} \\ &= \underline{r}_2(0) + r_2 \left[\frac{\underline{V}_2(0)}{\|\underline{V}_2(0)\|}, \underline{V}_2(0)^{\perp} \right] \begin{bmatrix} \cos(\eta_2) \\ \sin(\eta_2)\cos(\zeta_2) \\ \sin(\eta_2)\sin(\zeta_2) \end{bmatrix} \end{aligned} \tag{8}$$

where

$$\left[\frac{\underline{V}_i(0)}{\|\underline{V}_i(0)\|}, \underline{V}_i(0)^{\perp} \right] \in SO(3) \qquad i = 1, 2 \tag{9}$$

$$r_1 = \|\underline{r} - \underline{r}_1(0)\| \qquad r_2 = \|\underline{r} - \underline{r}_2(0)\|$$

$$\eta_1 = \cos^{-1}\left[\frac{\underline{V}_1(0) \cdot (\underline{r} - \underline{r}_1(0))}{\|\underline{V}_1(0)\| \, r_1} \right] \qquad \eta_2 = \cos^{-1}\left[\frac{\underline{V}_2(0) \cdot (\underline{r} - \underline{r}_2(0))}{\|\underline{V}_2(0)\| \, r_2} \right]$$

From equation 8, we also see that η_i is the angle between $\underline{V}_i(0)$ and $\underline{r} - \underline{r}_i(0)$, which is half the angle through which vehicle$_i$ has turned while on the circular trajectory from $\underline{r}_i(0)$ to $\underline{r}$.

Let T_i be the time it takes for vehicle$_i$ to travel along the circular trajectory from $\underline{r}_i(0)$ to $\underline{r}$. The circular distance that vehicle$_i$ travels is $\|\underline{V}_i(0)\| T_i$. The straight-line chord distance from $\underline{r}_i(0)$ to $\underline{r}$ is r_i. The ratio of circumferencial distance to chord distance is $\frac{\eta_i}{\sin(\eta_i)}$ where η_i is the half-angle. This gives us formulas for computing the arrival times as functions of r_i and η_i:

$$T_1 = \frac{r_1}{\|\underline{V}_1(0)\|} \frac{\eta_1}{\sin(\eta_1)} \tag{10}$$

$$T_2 = \frac{r_2}{\|\underline{V}_2(0)\|} \frac{\eta_2}{\sin(\eta_2)} \tag{11}$$

The intercept surface is then obtained by solving for all $\underline{r}$ that satisfy the equation: $T_1 = T_2$, i.e.

$$\frac{r_1}{r_2} = \frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \frac{\eta_2}{\sin(\eta_2)} \frac{\sin(\eta_1)}{\eta_1} \tag{12}$$

Equation 12 is a single equation in the three variables: $(r_1/r_2,\eta_1,\eta_2)$. Equations 8 and 12 give four scalar equations on the six variables (r_1,η_1,ζ_1), and (r_2,η_2,ζ_2). In either case there are two more variables than equations, so we get a two-dimensional solution set.

If we let:

$$k = \frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \frac{\eta_2}{\sin(\eta_2)} \frac{\sin(\eta_1)}{\eta_1} \tag{13}$$

then equation 12 can be written as $0 = F(\underline{r})$ where:

$$F(\underline{r}) = r_1 - k\, r_2 \tag{14}$$

3.2 Parameterization of the Intercept Surface

The data that describes the intercept surface can be represented as two vectors in a homogeneous space:

$$\text{data}_i = \begin{bmatrix} \underline{V}_i(0) \\ \underline{r}_i(0) \\ 1 \end{bmatrix} \quad i=1,2 \tag{15}$$

If we apply a Euclidean transformation that translates the origin to $\underline{r}_2(0)$, rotates the $\underline{V}_2(0)$ vector to the x axis, and rotates about this new x axis until the $\underline{r}_1(0) - \underline{r}_2(0)$ vector has no z component, then the data becomes:

$$\text{data}_{i_{\text{Euclidean}}} = \begin{bmatrix} R^T & 0_{3x3} & 0_{3x1} \\ 0_{3x3} & R^T & -R^T\underline{r}_2(0) \\ 0_{1x3} & 0_{1x3} & 1 \end{bmatrix} \text{data}_i$$

$$= \begin{bmatrix} R^T\,\underline{V}_1(0) \\ r_{12}\cos(\text{los}) \\ r_{12}\sin(\text{los}) \\ 0 \\ 1 \end{bmatrix} \text{ for } i=1 \quad \text{and} \quad \begin{bmatrix} V_2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \text{ for } i=2 \tag{16}$$

where los is the angle between the unit vector $\underline{1}_{V2}$ in the $\underline{V}_2(0)$ direction and the $\underline{1}_{r_{12}}$ unit vector in the $\underline{r}_1(0) - \underline{r}_2(0)$ direction.

$$\cos(\text{los}) = \underline{1}_{V2} \cdot \underline{1}_{r_{12}} \tag{17}$$

and

$$R = \left[\underline{1}_{V2}, \frac{\underline{1}_{V2} \times (\underline{1}_{v2} \times (\underline{r}_1(0) - \underline{r}_2(0)))}{\| \quad \|}, \frac{-\underline{1}_{v2} \times (\underline{r}_1(0) - \underline{r}_2(0))}{\| \quad \|} \right] \tag{18}$$

If we also uniformly scale distance by r_{12} and then scale time by V_2, we are left with a 4-

dimensional data set:

$$\text{data}_{i_{4\text{-dim}}} = \begin{bmatrix} (1/V_2)R^T & 0_{3x3} & 0_{3x1} \\ 0_{3x3} & (1/r_{12})R^T & -(1/r_{12})R^T\underline{r}_2(0) \\ 0_{1x3} & 0_{1x3} & 1 \end{bmatrix} \text{data}_i$$

$$= \begin{bmatrix} (V_1/V_2)\underline{1}_{V1} \\ \cos(\text{los}) \\ \sin(\text{los}) \\ 0 \\ 1 \end{bmatrix} \quad \text{for } i=1 \qquad \text{and} \qquad \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \text{for } i=2 \tag{19}$$

If we consider some additional condition on the shape of the intercept surface (such as when two surface components are disjoint) we can express this condition using the inequality:

$$V_1/V_2 \leq H(\underline{1}_{V1}, \text{los}) \quad \text{for some function H} \tag{20}$$

In the next section, we obtain a simple expression for when the surface components are disjoint, by finding another function, $\overline{H}(\text{los})$, which satisfies the inequality:

$$H(\underline{1}_{V1}, \text{los}) \leq \overline{H}(\text{los}) \qquad \text{for all } \underline{1}_{V1} \tag{21}$$

We can then ensure disjoint surface components whenever:

$$V_1/V_2 \leq \overline{H}(\text{los}) \tag{22}$$

To show the tightness of the bound, we also determine the value of $\underline{1}_{V1}$, that makes the above inequality become an equality.

3.3 Components of the Intercept Surface

The two points $\underline{r}_1(0)$ and $\underline{r}_2(0)$ are both on the intercept surface.

The scalar function $F(\underline{r})$, whose zero set is the intercept surface, is continuous everywhere except at the the two points $\underline{r}_1(0)$ and $\underline{r}_2(0)$.

We will determine conditions on the data that ensure that the solution component that includes the point $\underline{r}_1(0)$ is disjoint from the solution component that contains the point $\underline{r}_2(0)$.

Figures 2, 3, and 4 show cases where the components are separate, touching, and merged.

Figure 2: Intercept Surface With Two Separated Components

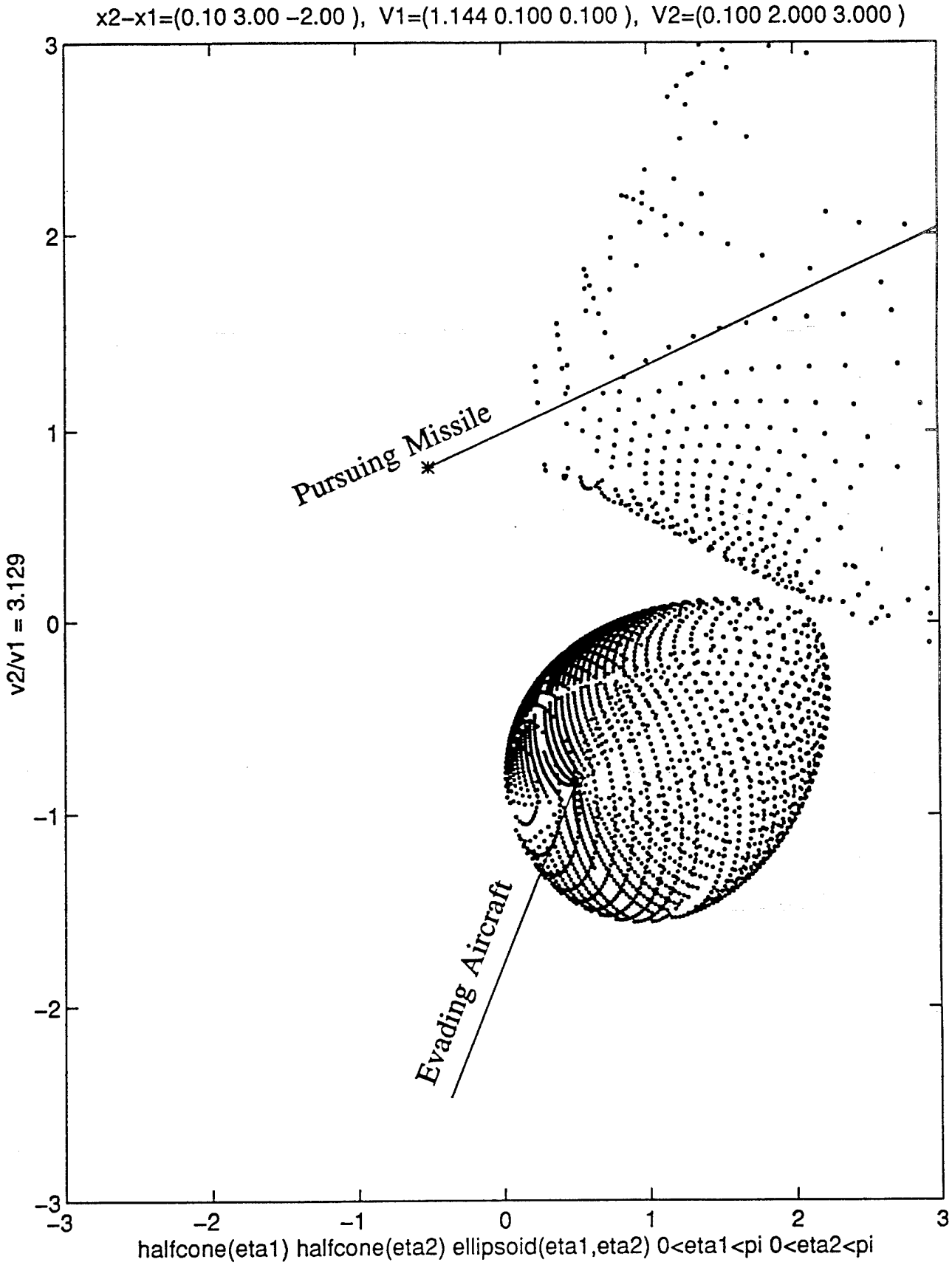


Figure 3: Intercept Surface With Two Touching Components

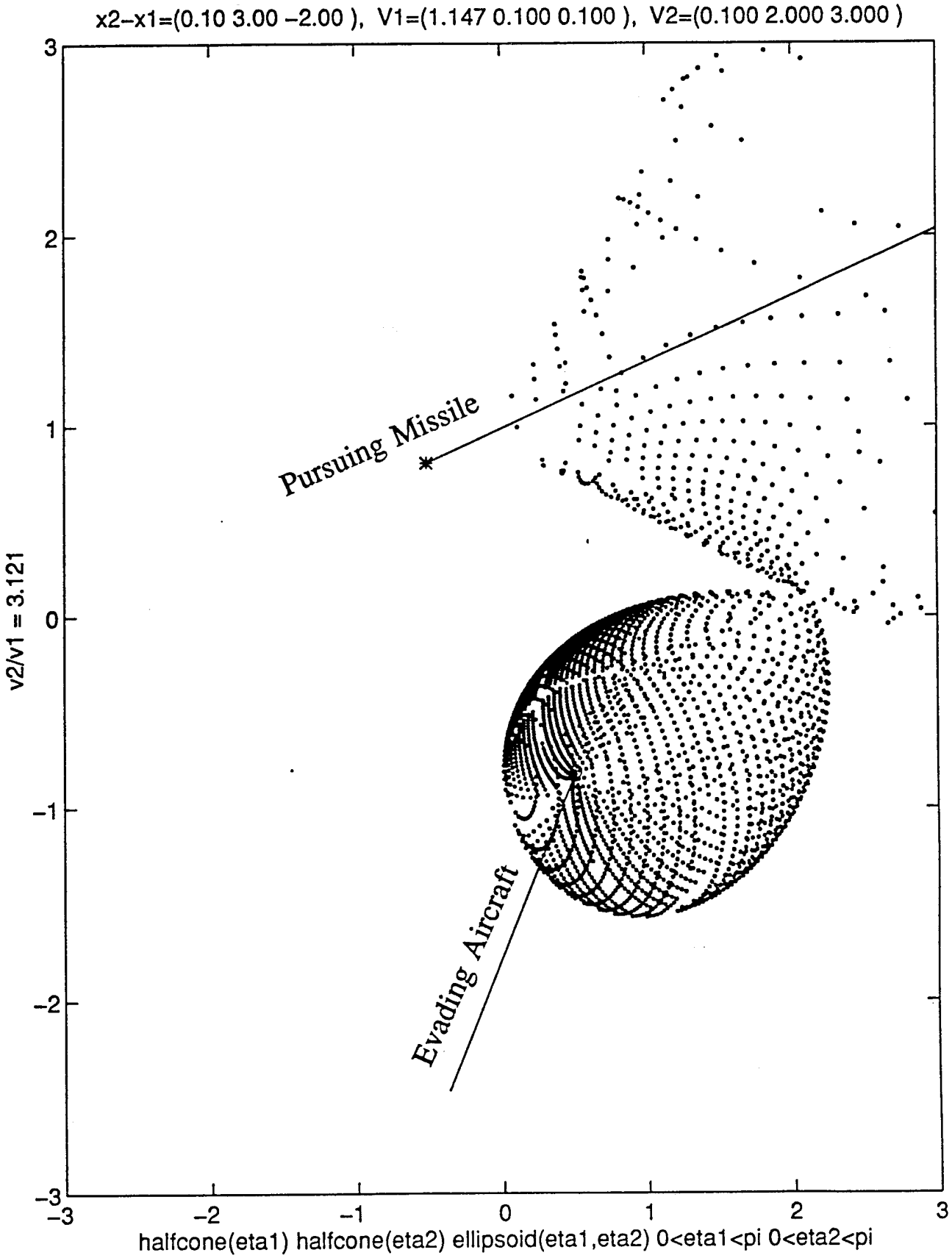
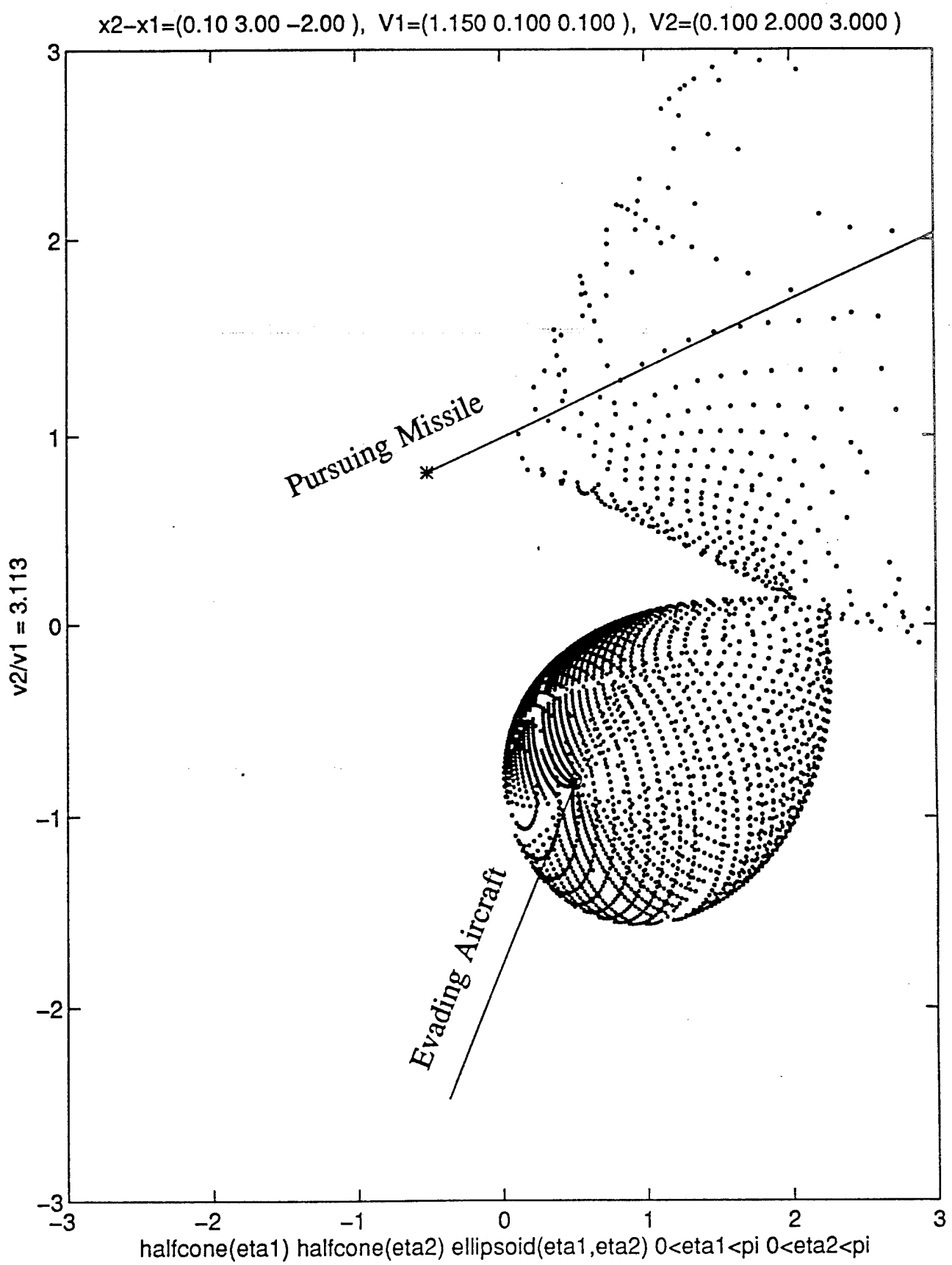


Figure 4: Intercept Surface With Merged Components



The point $\underline{r}_1(0)$ has $r_1 = 0$, so for all $0 \le \eta_{2_{const}} \le \pi$, it is contained in the solid ellipsoid:

$$r_1 \le \frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \frac{\eta_{2_{const}}}{\sin(\eta_{2_{const}})} r_2 \tag{23}$$

For all $0 \le \eta_{2_{const}} \le \pi$, the point $\underline{r}_2(0)$ is at the vertex of, and therefore is contained in, the solid half-cone:

$$\eta_2 \ge \eta_{2_{const}} \tag{24}$$

Theorem 1: For any fixed $0 \le \eta_{2_{const}} \le \pi$, the intercept surface is contained in the union of the solid ellipsoid and the solid cone described above.

To ensure that the solid half-cone $\eta_2 \le \eta_{2_{const}}$ does not intersect the solid ellipsoid, we must restrict the line-of-sight (los) angle between the cone axis, $\underline{V}_2(0)$, and the ellipsoid axis, $\underline{r}_2(0) - \underline{r}_1(0)$.

Define the line-of-sight angle:

$$los = \cos^{-1}\left[\frac{\underline{V}_2(0)^T}{\|\underline{V}_2(0)\|} \frac{[\underline{r}_1(0) - \underline{r}_2(0)]}{\|\underline{r}_1(0) - \underline{r}_2(0)\|} \right] \tag{25}$$

Theorem 2: If $0 \le v1/v2$ and $0 \le los \le \pi$ satisfy:

$$\frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \le \max_{0<los<\eta_{2_{const}}<\pi} \left[\frac{\sin(\eta_{2_{const}})}{\eta_{2_{const}}} \sin(\eta_{2_{const}} - los) \right]$$

$$= \frac{\sin(\eta_{2_{max}})}{\eta_{2_{max}}} \frac{1}{\sqrt{1 + \left[\frac{1}{\eta_{2_{max}}} - \frac{1}{\tan(\eta_{2_{max}})} \right]^2}} \tag{26}$$

where

$$\eta_{2_{max}} = (\pi - \sin^{-1}(v1/v2) + los)/2 \tag{27}$$

then the following equation has a solution with $0 \le los < \eta_{2_{const}} < \pi$

$$\frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} = \frac{\sin(\eta_{2_{const}})}{\eta_{2_{const}}} \sin(\eta_{2_{const}} - los) \tag{28}$$

For for the largest such $\eta_{2_{const}}$ solution, the solid ellipsoid and solid half-cone kiss. For values of $\eta_{2_{const}}$ just smaller than that solution, the solid ellipsoid and solid half-cone are disjoint, so the solution components that contain the points $\underline{r}_1(0)$ and $\underline{r}_2(0)$ are disjoint.

Conjecture: When the two solution components just touch, the max intercept time on the component that encloses the slower vehicle, occurs at the touch point.

Proof of Theorem 1: On the intercept surface, $F(\underline{r}) = 0$, yet in the region of space that is exterior to both the half-cone and the ellipsoid we have:

$$F(\underline{r}) = r_1 - k\, r_2$$

$$= r_1 - \frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \frac{\eta_2}{\sin(\eta_2)} \frac{\sin(\eta_1)}{\eta_1} r_2$$

$$\ge r_1 - \frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \frac{\eta_2}{\sin(\eta_2)} r_2 \qquad \text{for all } 0 \le \eta_1 \le \pi \tag{29}$$

$$> r_1 - \frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \frac{\eta_{2_{const}}}{\sin(\eta_{2_{const}})} r_2 \qquad \text{for } 0 \le \eta_2 < \eta_{2_{const}} \le \pi \quad \text{i.e. outside the solid half–cone}$$

> 0 outside the solid ellipsoid

So the region of space that is exterior to both the solid half-cone and the solid ellipsoid has

$T_1 > T_2$ so cannot contain points on the intercept surface (where $T_1 = T_2$).

For any point along the ray $\eta_1 = 0$ that is both inside the solid ellipsoid and inside the solid half-cone, the inequalities in the above proof are reversed and $T_1 < T_2$.

Proof of Theorem 2: As $\eta_{2_{const}}$ varies, the point where the solid half-cone and the solid ellipsoid first touch is contained in the plane that contains the cone axis, $\underline{V}_2(0)$ and the ellipsoid axis, $\underline{r}_2(0) - \underline{r}_1(0)$. In this plane, the cone becomes two rays based at $\underline{r}_2(0)$. The angles between these rays and the cone axis are $\pm\eta_{2_{const}}$. Let $\underline{r}$ be a vector along the ray with angle $+\eta_{2_{const}}$. This ray can intersect the planar ellipse at up to two points. When the two solutions coalesce, the cone and ellipse are tangent to each other.

The law of cosines applied to the triangle with vertices $\underline{r}_1(0)$, $\underline{r}_2(0)$, and $\underline{r}$ can be written as:

$$r_1^2 = r_2^2 + r_{12}^2 - 2\, r_2\, r_{12} \cos(\eta_2 - los) \tag{30}$$

The boundary of the solid ellipsoid is given by:

$$r_1 = \frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \frac{\eta_{2_{const}}}{\sin(\eta_{2_{const}})}\, r_2 \tag{31}$$

Eliminating r_1 in the above two equations gives:

$$0 = \left[1 - \left[\frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \frac{\eta_{2_{const}}}{\sin(\eta_{2_{const}})}\right]^2\right] r_2^2 + r_{12}^2 - 2\, r_2\, r_{12} \cos(\eta_{2_{const}} - los) \tag{32}$$

This quadratic equation has two solutions that give the normalized distance r_2 along the cone ray.

This quadratic equation has roots:

$$\frac{r_2}{r_{12}} = \frac{\cos(\eta_{2_{const}} - los) \pm \sqrt{\cos^2(\eta_{2_{const}} - los) - (1 - [\frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \frac{\eta_{2_{const}}}{\sin(\eta_{2_{const}})}]^2)}}{(1 - [\frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \frac{\eta_{2_{const}}}{\sin(\eta_{2_{const}})}]^2)} \tag{33}$$

The two roots coalesce when

$$\cos(\eta_{2_{const}} - los) = \pm \sqrt{(1 - [\frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \frac{\eta_{2_{const}}}{\sin(\eta_{2_{const}})}]^2)} \tag{34}$$

or

$$|\sin(\eta_{2_{const}} - los)| = \frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \frac{\eta_{2_{const}}}{\sin(\eta_{2_{const}})} \qquad \text{solve for } \eta_{2_{const}} \tag{35}$$

or

$$\frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} = \frac{\sin(\eta_{2_{const}})}{\eta_{2_{const}}} |\sin(\eta_{2_{const}} - los)| \tag{36}$$

To obtain the largest value of $\frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|}$ for which solutions to equation 36 exist, note that

$$(.99 - \sin(\frac{los}{2}))*.724 < \max_{0<los<\eta_{2_{const}}<\pi} \left[\frac{\sin(\eta_{2_{const}})}{\eta_{2_{const}}} \sin(\eta_{2_{const}} - los) \right] < (1.03 - \sin(los/2))*.724$$

and the maximum occurs at the value of $\eta_{2_{const}}$ given by:

$$\sin(2\eta_{2_{const}} - los) = v1/v2 \qquad 2 \text{ branches for } \sin^{-1} \tag{38}$$

We need $\eta_{2_{const}}$ in the range $0 \le los < \eta_{2_{const}} < \pi$ so we choose the branch with the largest value of $\eta_{2_{const}}$, e.g.

$$\eta_{2_{const}} = (\pi - asin(v1/v2) + los)/2 \tag{39}$$

The above bounds imply that:

$$\text{equation 36 has a solution} \quad \text{if} \quad \frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \le .724*(.99 - \sin(los/2)) \tag{40}$$

while

$$\text{equation 36 has no solution} \quad \text{if} \quad \frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \ge .724*(1.03 - \sin(los/2)) \tag{41}$$

After solving equation 36 for $\eta_{2_{const}}$, the location of the point where the solid ellipsoid and the solid half-cone just touch can be determined by:

$$\frac{r_2}{r_{12}} = \frac{1}{\cos(\eta_{2_{const}} - los)} \tag{42}$$

$$\frac{r_1}{r_{12}} = \tan(\eta_{2_{const}} - los) \tag{43}$$

Examples:

For fixed values of los, we can plot equation 36 to find the value of $\eta_{2_{const}} > los$ that gives the largest value of $\frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|}$.

$$los = 1.57 \text{ rad} = 90 \text{ deg} \quad \text{gives} \quad \min_{\eta_{2_{const}}} \left[\frac{\|\underline{V}_2(0)\|}{\|\underline{V}_1(0)\|} \right] = 4.60 \quad \text{at} \quad \eta_{2_{const}} = 2.25 \tag{44}$$

$$los = 1.00 \text{ rad} = 57 \text{ deg} \quad \text{gives} \quad \min_{\eta_{2_{const}}} \frac{\|\underline{V}_2(0)\|}{\|\underline{V}_1(0)\|} = 2.56 \quad \text{at} \quad \eta_{2_{const}} = 1.87 \tag{45}$$

Tight Bound

The inequality in theorem 2 becomes an equality when

$$V_1/V_2 = \max_{0<\eta_{2_{const}}<los<\pi} \left[\frac{\sin(\eta_{2_{const}})}{\eta_{2_{const}}} \sin(\eta_{2_{const}} - los) \right] \approx .742\ (1 - \sin(los/2)) \qquad (46)$$

which occurs at

$$\eta_{2_{const}} = (\pi - \sin^{-1}(v1/v2) + los)/2 \qquad (47)$$

The inequalities in theorem 1 become equalities when $\eta_1 = 0$ and $\eta_2 = \eta_{2_{const}}$ (solution of equation in theorem 2). This implies that the triangle with sides of length r_1, r_2, and r_{12} is a right triangle, since:

$$\sin(\eta_{2_{const}} - los) = \frac{\underline{V}_1(0)}{\underline{V}_2(0)} \frac{\eta_{2_{const}}}{\sin(\eta_{2_{const}})} = k = \frac{r_1}{r_2} \qquad \text{when} \quad \eta_1 = 0, \quad \eta_2 = \eta_{2_{const}} \qquad (48)$$

To orient the $\underline{V}_1(0)$ vector in the direction that results in $\eta_1 = 0$ at the point where the cone and ellipsoid meet, we need:

1) $\underline{V}_1(0)$ must be in the plane spanned by the cone axis, $\underline{V}_2(0)$, and the ellipsoid axis, $\underline{r}_2(0) - \underline{r}_1(0)$.

2) $\underline{V}_1(0)$ must be perpendicular to the ellipsoid axis, $\underline{r}_2(0) - \underline{r}_1(0)$.

This can be done by pointing $\underline{V}_1(0)$ along the direction:

$$(\underline{r}_2(0) - \underline{r}_1(0)) \times [(\underline{r}_2(0) - \underline{r}_1(0)) \times \underline{V}_2(0)] \qquad (49)$$

3.4 Singularities on the Intercept Surface

The two points $\underline{r}_1(0)$ and $\underline{r}_2(0)$ are both on the intercept surface.

The function $F(\underline{r})$ that defines the intercept surface is continuous everywhere except at the the two points $\underline{r}_1(0)$ and $\underline{r}_2(0)$.

To determine if the intercept surface has any other singularities, we can examine the gradient of the function $F(\underline{r})$ to see if it is bounded away from zero. Figure 5 shows a case where the intercept surface has a singularity.

The following notation is useful in deriving the gradient of F:

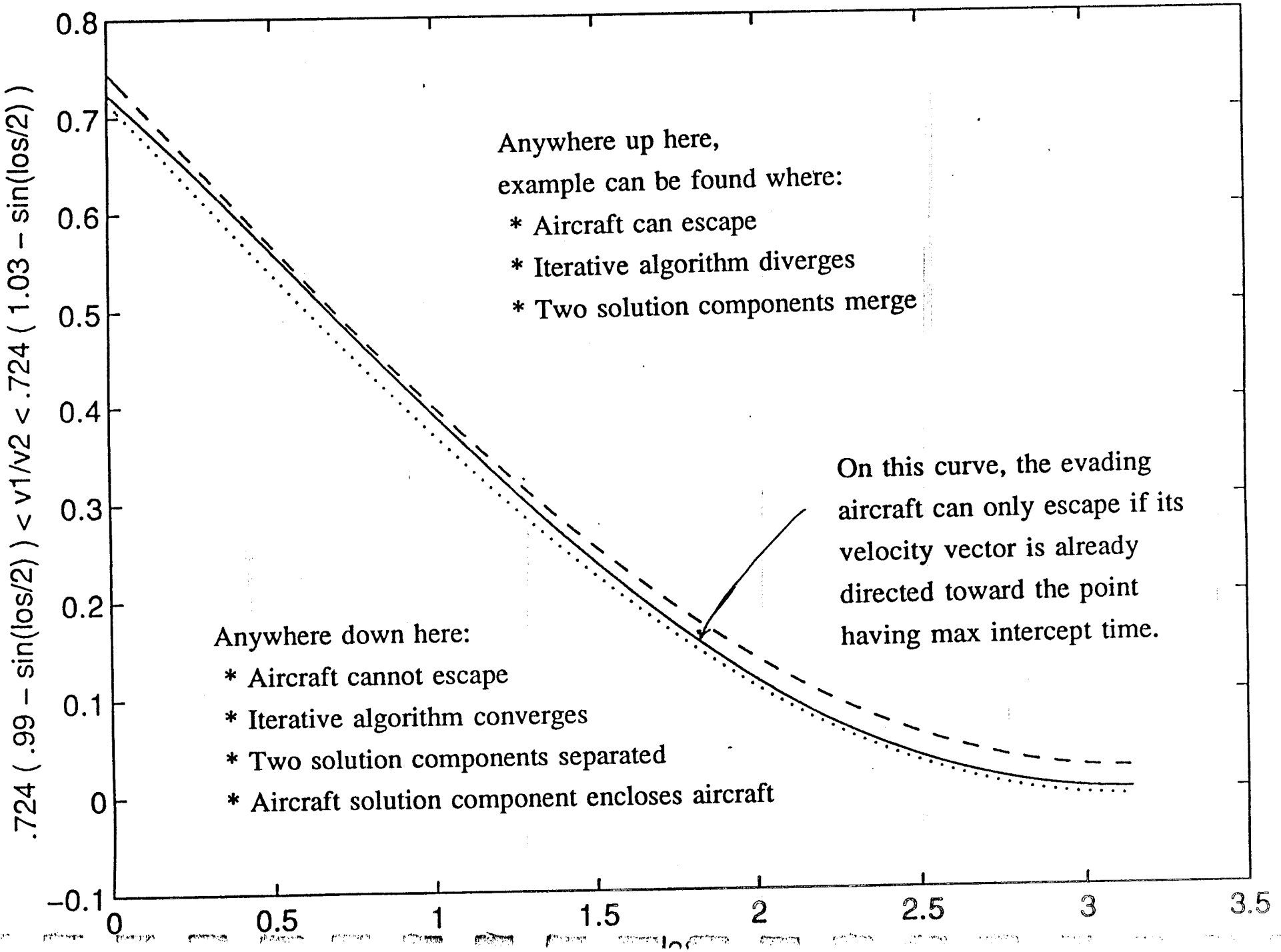
$$\underline{1}_{r_i} = \frac{\underline{r} - \underline{r}_i(0)}{r_i} \qquad i = 1,2 \qquad \text{unit vectors} \tag{50}$$

$$\underline{h}_i = \frac{(\underline{r} - \underline{r}_i(0)) \times (\underline{r} - \underline{r}_i(0)) \times \underline{V}_i(0)}{\|(\underline{r} - \underline{r}_i(0)) \times (\underline{r} - \underline{r}_i(0)) \times \underline{V}_i(0)\|} \qquad i=1,2 \tag{51}$$

Note that for each i, $\underline{1}_{r_i}$ and $\underline{h}_i$ are unit vectors and are orthogonal to each other, so we can define the following rotation matrices:

$$C_i = [\underline{1}_{r_i}, \underline{h}_i, \underline{1}_{r_i} \times \underline{h}_i] \qquad i=1,2 \tag{52}$$

Figure 5: max v1/v2 that satisfies separate solution component bound



The normal to the surface is given by the gradient of the function that defines the surface.

$$\begin{aligned}\frac{dF}{d\underline{r}} &= \frac{dr_1}{d\underline{r}} - k\frac{dr_2}{d\underline{r}} - \frac{dk}{d\underline{r}} r_2 \\ &= \underline{1}_{r_1} - k\,\underline{1}_{r_2} - r_2\frac{dk}{d\underline{r}}\end{aligned} \tag{53}$$

To evaluate $\frac{dk}{d\underline{r}}$, recall that

$$k = \frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|}\frac{\sin(\eta_1)}{\eta_1}\frac{\eta_2}{\sin(\eta_2)} \tag{54}$$

$$\frac{d}{d\eta_1}\left(\frac{\sin(\eta_1)}{\eta_1}\right) = -\left(\frac{\sin(\eta_1)}{\eta_1}\right)\left(\frac{1}{\eta_1} - \frac{1}{\tan(\eta_1)}\right) \tag{55}$$

$$\frac{d}{d\eta_2}\left(\frac{\eta_2}{\sin(\eta_2)}\right) = \left(\frac{\eta_2}{\sin(\eta_2)}\right)\left(\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)}\right) \tag{56}$$

Combining the above 3 equations gives:

$$\begin{aligned}\frac{dk}{d\underline{r}} &= -\frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|}\frac{\sin(\eta_1)}{\eta_1}\frac{\eta_2}{\sin(\eta_2)}\left[\left(\frac{1}{\eta_1} - \frac{1}{\tan(\eta_1)}\right)\frac{d\eta_1}{d\underline{r}} - \left(\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)}\right)\frac{d\eta_2}{d\underline{r}}\right] \\ &= -k\left[\left(\frac{1}{\eta_1} - \frac{1}{\tan(\eta_1)}\right)\frac{d\eta_1}{d\underline{r}} - \left(\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)}\right)\frac{d\eta_2}{d\underline{r}}\right]\end{aligned} \tag{57}$$

The angle from the vector $\underline{V}_i(0)$ and $\underline{r} - \underline{r}_i(0)$ is η_i, so $\frac{d\eta_i}{d\underline{r}}$ is a vector of length $1/r_i$, perpendicular to $\underline{r} - \underline{r}_i(0)$, in the plane spanned by $\underline{V}_i(0)$ and $\underline{r} - \underline{r}_i(0)$. The unit vector $\underline{h}_i$, defined above, is in the appropriate direction, so

$$\frac{d\eta_i}{d\underline{r}} = \frac{1}{r_i}\underline{h}_i \tag{58}$$

Combining the above equations [and noting that $\frac{r_1}{r_2} = k$ when $F(\underline{r}) = 0$] gives:

$$\frac{dF}{d\underline{r}} = \left[\underline{1}_{r_1} + \left(\frac{1}{\eta_1} - \frac{1}{\tan(\eta_1)}\right) \underline{h}_1 \right] - k \left[\underline{1}_{r_2} + \left(\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)}\right) \underline{h}_2 \right] =$$

$$C_1 \begin{bmatrix} 1 \\ \frac{1}{\eta_1} - \frac{1}{\tan(\eta_1)} \\ 0 \end{bmatrix} - k\, C_2 \begin{bmatrix} 1 \\ \frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)} \\ 0 \end{bmatrix} \tag{59}$$

Applying the triangle inequality to the above equation, we get:

$$\left\| \frac{dF}{d\underline{r}} \right\| \geq \sqrt{1 + \left(\frac{1}{\eta_1} - \frac{1}{\tan(\eta_1)}\right)^2} - |k| \cdot \sqrt{1 + \left(\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)}\right)^2}$$

$$\geq 1 - \left[\frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \frac{\eta_2}{\sin(\eta_2)} \right] \sqrt{1 + \left(\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)}\right)^2} \tag{60}$$

The gradient cannot be zero in regions of space where η_2 satisfies:

$$\frac{\|\underline{V}_2(0)\|}{\|\underline{V}_1(0)\|} > \frac{\eta_2}{\sin(\eta_2)} \sqrt{1 + \left(\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)}\right)^2} \tag{61}$$

This represents a half-cone region in front of the missile, up to an angle of η_2 from the velocity vector $\underline{V}_2(0)$. This means that the intercept surface cannot have any singularities in that region of space, except at the point $\underline{r} = \underline{r}_1(0)$ where the function itself is discontinuous. The function is also discontinuous at the point $\underline{r} = \underline{r}_2(0)$, but that point is outside the η_2 bound given above.

For example, if $\frac{\|\underline{V}_2(0)\|}{\|\underline{V}_1(0)\|} = 3$ then the gradient cannot be zero in regions where $\eta_2 < 1.97\text{rad} = 112\text{deg}$.

All possible singular points:

In an earlier section, we showed that the 12-dimensional set of data: $\underline{r}_1(0)$, $\underline{r}_2(0)$, $\underline{V}_1(0)$, $\underline{V}_2(0)$ could be reduced to a 4-dimensional set by modding out by the 6-dimensional Euclidean group of motions, scaling distance, and scaling time. For each point in the remaining 4-dimensional data set, we get a 2-dimensional solution set for $\underline{r} \in \mathbf{R}^3$. The vector equation, $d\underline{F}/d\underline{r} = \underline{0}$, is equivalent to 3 scalar equations, which give fixed points on the 2-dimensional solution set, and put a single condition on the 4-dimensional data set. Therefore, there is a 3-dimensional set of degenerate data, that results in the solution surface having a singular point. This 3-dimensional set of degenerate data can be parameterized by the following rotation matrix:

$$C = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} = C_1^T C_2 \in SO(3) \tag{62}$$

The solution set is singular when

$$\underline{0} = \frac{d\underline{F}}{d\underline{r}} \tag{63}$$

i.e.,

$$\begin{bmatrix} \frac{1}{\eta_1} - \frac{1}{\tan(\eta_1)} \\ 1 \\ 0 \end{bmatrix} = k\,(C_1^T C_2) \begin{bmatrix} \frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)} \\ 1 \\ 0 \end{bmatrix} \tag{64}$$

We can solve these three equations for k, $\frac{1}{\eta_1} - \frac{1}{\tan(\eta_1)}$ and $\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)}$.

$$\frac{1}{\eta_1} - \frac{1}{\tan(\eta_1)} = \frac{-c_{31}}{c_{32}} \tag{65}$$

$$\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)} = \frac{-c_{13}}{c_{13}} \tag{66}$$

$$k = \frac{\sqrt{1 + (\frac{1}{\eta_1} - \frac{1}{\tan(\eta_1)})^2}}{\sqrt{1 + (\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)})^2}} = \frac{\sqrt{1 + (\frac{c_{31}}{c_{32}})^2}}{\sqrt{1 + (\frac{c_{13}}{c_{23}})^2}} \tag{67}$$

The above three equations can then be used to solve for η_1, η_2, and $\|\underline{V}_1(0)\|/\|\underline{V}_2(0)\|$ as functions of C. These values of η_1 and η_2, give a point on the 2-dimensional solution set. The 4-dimensional data set was parameterized by $\|\underline{V}_1(0)\|/\|\underline{V}_2(0)\|$, los-angle, and $\underline{1}_{V_1} \in S^2$.

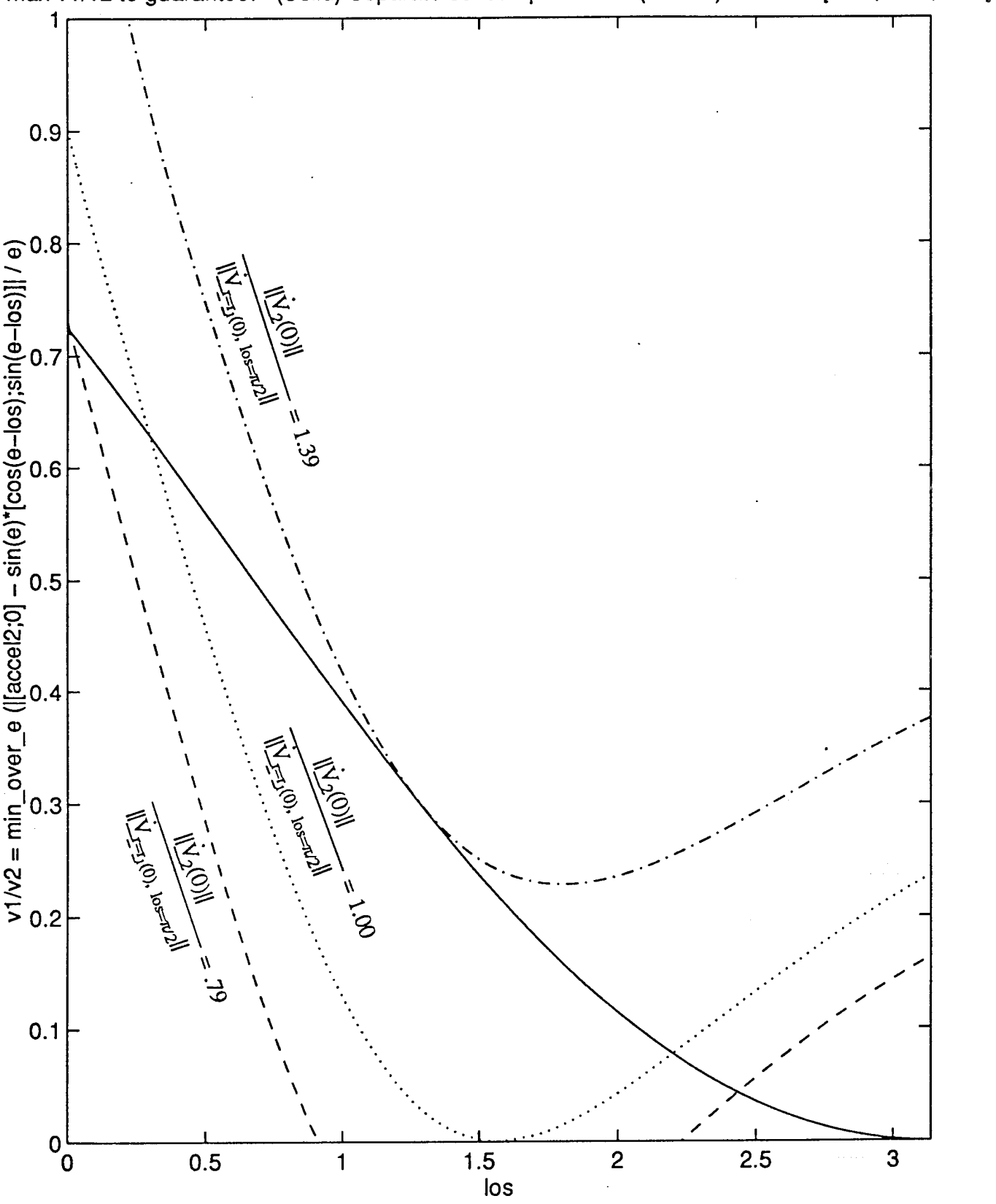
3.5 Acceleration Constraints for the Pursuer

In this subsection, we will look at two things. First we will derive a bound of the form:

$$v1/v2 < \text{function(los,accel\_limit)} \tag{68}$$

that guarantees that the pursuer will not violate its acceleration constraint anywhere on the selected component of the intercept surface. Second, we will derive a set of nonlinear equations whose solution gives initial conditions and a point on the intercept surface where the pursuer's acceleration constraint is just reached. Figures 6-9 show acceleration constraints on the intercept surface.

Figure 6: Acceleration Limits On Pursuing Missile



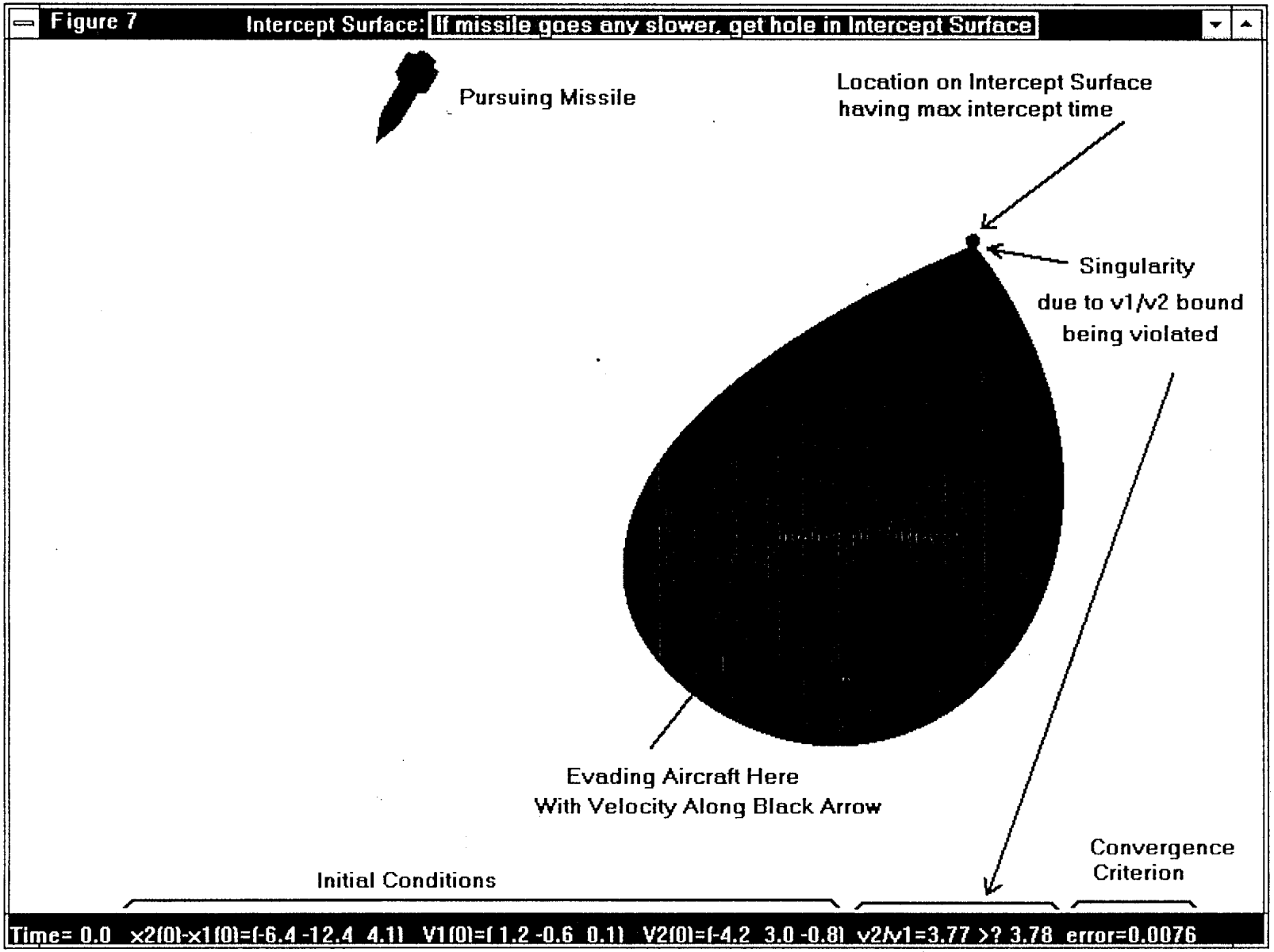


Figure 7
Intercept Surface: If missile goes any slower, get hole in Intercept Surface
Pursuing Missile
Location on Intercept Surface
having max intercept time
Singularity
due to v1/v2 bound
being violated
Evading Aircraft Here
With Velocity Along Black Arrow
Convergence
Criterion
Initial Conditions
Time= 0.0 x2(0)-x1(0)=(-6.4 -12.4 4.1) V1(0)=(1.2 -0.6 0.1) V2(0)=(-4.2 3.0 -0.8) v2/v1=3.77 >? 3.78 error=0.0076

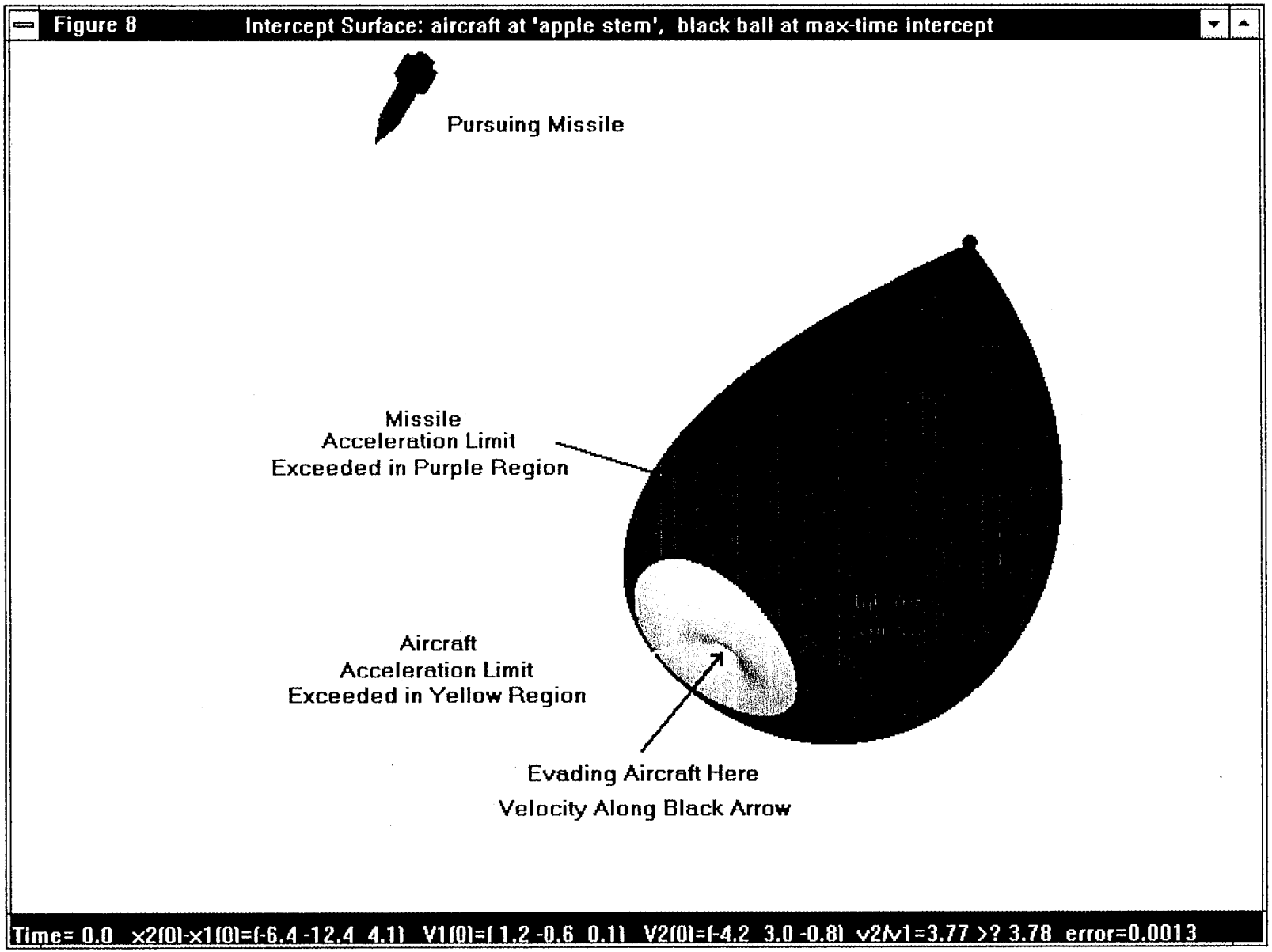


Figure 8
Intercept Surface: aircraft at 'apple stem', black ball at max-time intercept
Pursuing Missile
Missile
Acceleration Limit
Exceeded in Purple Region
Aircraft
Acceleration Limit
Exceeded in Yellow Region
Evading Aircraft Here
Velocity Along Black Arrow
Time= 0.0 x2(0)-x1(0)=(-6.4 -12.4 4.1) V1(0)=(1.2 -0.6 0.1) V2(0)=(-4.2 3.0 -0.8) v2/v1=3.77 >? 3.78 error=0.0013

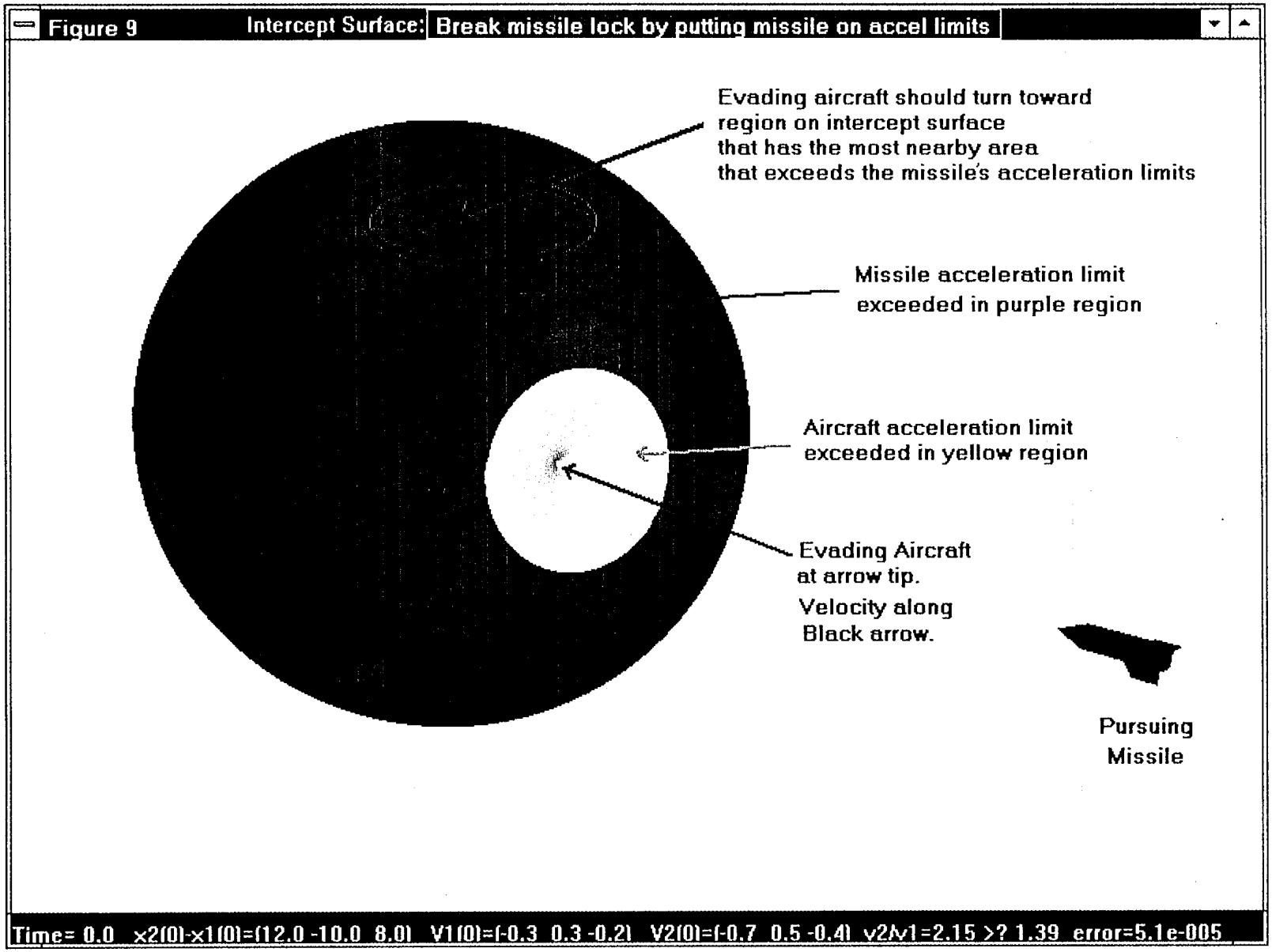


Figure 9
Intercept Surface: Break missile lock by putting missile on accel limits
Evading aircraft should turn toward
region on intercept surface
that has the most nearby area
that exceeds the missile's acceleration limits
Missile acceleration limit
exceeded in purple region
Aircraft acceleration limit
exceeded in yellow region
Evading Aircraft
at arrow tip.
Velocity along
Black arrow.
Pursuing
Missile
Time= 0.0 x2(0)-x1(0)=(12.0 -10.0 8.0) V1(0)=(-0.3 0.3 -0.2) V2(0)=(-0.7 0.5 -0.4) v2/v1=2.15 >? 1.39 error=5.1e-005

To determine the bound v1/v2 < function(los,accel_limit), we start with the equation that defines the intercept surface:

$$\frac{\sin(\eta_1)}{\eta_1}\, v1/v2 = (r1/r2)\frac{\sin(\eta_2)}{\eta_2} \tag{69}$$

and the equation that defines the surface of constant acceleration for the pursuer, vehicle 2:

$$\|\dot{\underline{V}}_2(0)\| = \frac{2\|\underline{V}_2(0)\|^2 \sin(\eta_2)}{r_2} \tag{70}$$

In equation 69, we can rewrite r1 as:

$$\begin{aligned} r_1 &= \|(\underline{r} - \underline{r}_1(0))\| \\ &= \|(\dot{\underline{r}}_1(0) - \underline{r}_2(0)) - (\underline{r} - \underline{r}_2(0))\| \\ &= \|(\underline{r}_1(0) - \underline{r}_2(0)) - [\underline{1}_{v2},\, \underline{1}_{v2}{}^{\perp}]\, r_2\, \underline{1}_{\eta_2,\zeta_2}\| \\ &= \|[\underline{1}_{v2},\, \underline{1}_{v2}{}^{\perp}]^T\, (\underline{r}_1(0) - \underline{r}_2(0)) - r_2\, \underline{1}_{\eta_2,\zeta_2}\| \\ &= \|r_{12} \begin{bmatrix} \cos(los) \\ \sin(los) \\ 0 \end{bmatrix} - r_2 \begin{bmatrix} \cos(\eta_2) \\ \sin(\eta_2)\cos(\zeta_2) \\ \sin(\eta_2)\sin(\zeta_2) \end{bmatrix}\| \end{aligned} \tag{71}$$

Putting this expression for r_1 into equation 69 gives:

$$\frac{\sin(\eta_1)}{\eta_1}\, v1/v2 = \|(r_{12}/r_2) \begin{bmatrix} \cos(los) \\ \sin(los) \\ 0 \end{bmatrix} - \begin{bmatrix} \cos(\eta_2) \\ \sin(\eta_2)\cos(\zeta_2) \\ \sin(\eta_2)\sin(\zeta_2) \end{bmatrix}\| \frac{\sin(\eta_2)}{\eta_2} \tag{72}$$

Solving equation 70 for r_2 and putting that into equation 72 gives:

$$\frac{\sin(\eta_1)}{\eta_1}\, v1/v2 = \|\frac{\|\dot{\underline{V}}_2(0)\|}{\|\dot{\underline{V}}_{2,\, \underline{r}=\underline{r}_1(0),\, \eta_2=\pi/2}\|} \begin{bmatrix} \cos(los) \\ \sin(los) \\ 0 \end{bmatrix} - \sin(\eta_2) \begin{bmatrix} \cos(\eta_2) \\ \sin(\eta_2)\cos(\zeta_2) \\ \sin(\eta_2)\sin(\zeta_2) \end{bmatrix}\| \frac{1}{\eta_2} \tag{73}$$

where

$$\|\dot{\underline{V}}_{2,\ \underline{r}=\underline{r}_1(0),\ \eta_2=\pi/2}\| = \frac{2\|\underline{V}_2(0)\|^2}{r_{12}} \tag{74}$$

The right-hand side of equation 73 is minimized when $\zeta_2 = 0$. If we consider examples where the $\underline{V}_1(0)$ vector is rotated till $\eta_1 = 0$ at the point where the intercept surface and the acceleration constraint surface just touch, we get:

$$(v1/v2)_{touch} = \min_{0<\eta_2<\pi} \left\| \frac{\|\dot{\underline{V}}_2(0)\|}{\|\dot{\underline{V}}_{2,\ \eta_2=\pi/2}\|} \begin{bmatrix}\cos(los)\\ \sin(los)\end{bmatrix} - \sin(\eta_2) \begin{bmatrix}\cos(\eta_2)\\ \sin(\eta_2)\end{bmatrix} \right\| \frac{1}{\eta_2} \tag{75}$$

Actually, the min should be taken over the restricted interval:

$$\min_{0<\eta_2<(3\ los\ +\ \pi)/4}$$

to ensure that we are finding the intersection of the constant $accel_2$ surface with the component of the intercept surface that contains $\underline{r}_1(0)$.

Equation 75 gives the value of v1/v2 that corresponds to the intercept surface just touching the accel constraint surface. Larger values of v1/v2 could result in either larger or smaller values of max acceleration. To eliminate cases where smaller values of v1/v2 result in larger accelerations, we can set

$$(v1/v2)_{accel<constraint} = \begin{cases} 0 \quad \text{whenever} \quad \sin(\eta_{2_{touch}}) > \dfrac{\|\dot{\underline{V}}_2\|}{\left[\dfrac{2\|\underline{V}_2(0)\|^2}{r_{12}}\right]} \\ (v1/v2)_{touch} \quad \text{whenever} \quad \sin(\eta_{2_{touch}}) < \dfrac{\|\dot{\underline{V}}_2\|}{\left[\dfrac{2\|\underline{V}_2(0)\|^2}{r_{12}}\right]} \end{cases} \tag{76}$$

Examples where acceleration constraint is just met

Constraining a vehicle to a fixed acceleration magnitude, means that it will be on a fixed radius circular path (tangent to its velocity vector). The set of all such circular paths forms a 2-dimensional surface. The fixed value of acceleration that just causes this surface to touch the intercept surface is the max acceleration on the intercept surface. To solve for the intersection point of these two tangential surfaces, we must set the gradient of the constant acceleration surface, equal to a constant times the gradient of the intercept surface. The magnitude of the acceleration on the two surfaces must also be set equal. This gives 3 equations to solve for the magnitude of the acceleration and the point on the two-dimensional intercept surface where that acceleration occurs.

The acceleration of vehicle i is given by:

$$\underline{\text{accel}}_i = [\underline{1}_{r_i}, \underline{h}_i] \begin{bmatrix} \sin(\eta_i) \\ \cos(\eta_i) \end{bmatrix} \frac{2\|V_i(0)\|^2 \sin(\eta_i)}{r_i} \tag{77}$$

so the magnitude of the acceleration is constant on the 2-dim surface defined by the function:

$$\|\underline{\text{accel}}_i\| = \frac{2\|V_i(0)\|^2 \sin(\eta_i)}{r_i} \tag{78}$$

For fixed values of $\|\underline{\text{accel}}_i\|$, the gradient of this function is:

$$\frac{d}{d\underline{r}} \left[\frac{2\|V_i(0)\|^2 \sin(\eta_i)}{r_i} \right] = [\underline{1}_{r_i}, \underline{h}_i] \begin{bmatrix} -\sin(\eta_i) \\ \cos(\eta_i) \end{bmatrix} \frac{2\|V_i(0)\|^2}{r_i^2} = C_i \begin{bmatrix} -\sin(\eta_i) \\ \cos(\eta_i) \\ 0 \end{bmatrix} \frac{2\|V_i(0)\|^2}{r_i^2} \tag{79}$$

Setting a scalar times $\frac{d\|\underline{\text{accel}}_2\|}{d\underline{r}}$ equal to $\frac{dF}{d\underline{r}}$ gives three equations for that scalar and the touch point on the 2-dimensional intercept surface:

$$C_2 \begin{bmatrix} -\sin(\eta_2) \\ \cos(\eta_2) \\ 0 \end{bmatrix} a_c = C_1 \begin{bmatrix} \frac{1}{\eta_1} - \frac{1}{\tan(\eta_1)} \\ 0 \end{bmatrix} - C_2 \begin{bmatrix} \frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)} \\ 0 \end{bmatrix} k \tag{80}$$

or

$$\begin{bmatrix} \frac{1}{\eta_1} - \frac{1}{\tan(\eta_1)} \\ 0 \end{bmatrix} = (C_1^T C_2) \begin{bmatrix} 1 & -\sin(\eta_2) \\ \frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)} & \cos(\eta_2) \\ 0 & 0 \end{bmatrix} \begin{bmatrix} k \\ a_c \end{bmatrix} \tag{81}$$

Solving for $\frac{1}{\eta_1} - \frac{1}{\tan(\eta_1)}$, k, and a_c gives:

$$\frac{1}{\eta_1} - \frac{1}{\tan(\eta_1)} = \frac{-c_{13}}{c_{23}} \tag{82}$$

$$\begin{bmatrix} k \\ a_c \end{bmatrix} = \frac{\eta_2}{\sin(\eta_2)} \begin{bmatrix} \cos(\eta_2) & \sin(\eta_2) \\ -[\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)}] & 1 \end{bmatrix} \begin{bmatrix} c_{11} & c_{21} \\ c_{12} & c_{22} \end{bmatrix} \begin{bmatrix} \frac{1}{\eta_1} - \frac{1}{\tan(\eta_1)} \\ 1 \end{bmatrix} \tag{83}$$

$$= \frac{\eta_2}{\sin(\eta_2)} \begin{bmatrix} \cos(\eta_2) & \sin(\eta_2) \\ -[\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)}] & 1 \end{bmatrix} \begin{bmatrix} c_{11} - c_{21} \frac{c_{13}}{c_{23}} \\ c_{12} - c_{22} \frac{c_{13}}{c_{23}} \end{bmatrix}$$

Since

$$k = \frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|} \frac{\sin(\eta_1)}{\eta_1} \frac{\eta_2}{\sin(\eta_2)} \tag{85}$$

we can combine equation 85 with the top half of equation 83 (for k) to get:

$$\frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|}\,\frac{\sin(\eta_1)}{\eta_1} = [\cos(\eta_2),\ \sin(\eta_2)] \begin{bmatrix} c_{11} - c_{21}\,\dfrac{c_{13}}{c_{23}} \\ c_{12} - c_{22}\,\dfrac{c_{13}}{c_{23}} \end{bmatrix} \tag{86}$$

In the next two sections, we will present both iterative and closed-form techniques for calculating points on the two-dimensional surface defined by the scalar equation $F(\underline{r}) = 0$.

4. Iterative Algorithm for Computing the Intercept Surface

This section defines an iterative algorithm for computing the radius, r_1, along each fixed unit vector, $\underline{1}_{\eta_1,\zeta_1}$, such that the vector $\underline{r} = \underline{r}_1(0) + r_1 \underline{1}_{\eta_1,\zeta_1}$ is on the intercept surface.

From equations 8 and 9,

$$r_1 = \|\underline{r} - \underline{r}_1(0)\| \qquad r_2 = \|\underline{r} - \underline{r}_2(0)\| \tag{87}$$

Using the law of cosines on the triangle with vertices $\underline{r}_1(0)$, $\underline{r}_2(0)$ and $\underline{r}$, we can write r_2 as a function of r_1, η_1, ζ_1:

$$r_2^2 = r_1^2 - 2\cos(\phi)\, r_1 \|\underline{r}_1(0) - \underline{r}_2(0)\| + \|\underline{r}_1(0) - \underline{r}_2(0)\|^2 \tag{88}$$

where ϕ is the angle between the vectors $\underline{r}_1(0) - \underline{r}_2(0)$ and $\underline{r} - \underline{r}_1(0)$. This angle is fixed during the iteration, since the unit vector along $\underline{r} - \underline{r}_1(0)$ is only a function of η_1 and ζ_1.

From equation 8, we also see that η_2 is the angle between $\underline{V}_2(0)$ and $\underline{r} - \underline{r}_2(0)$, so η_2 can be written as a function of r_1, η_1, ζ_1:

$$\eta_2 = \cos^{-1}\left[\frac{\underline{V}_2(0) \cdot (\underline{r} - \underline{r}_2(0))}{\|\underline{V}_2(0)\| \, \|\underline{r} - \underline{r}_2(0)\|}\right] \tag{89}$$

where

$$\underline{r} - \underline{r}_2(0) = [\underline{r}_1(0) - \underline{r}_2(0)] + r_1 \left[\frac{\underline{V}_1(0)}{\|\underline{V}_1(0)\|}, \underline{V}_1(0)^{\perp}\right] \begin{bmatrix} \cos(\eta_1) \\ \sin(\eta_1)\cos(\zeta_1) \\ \sin(\eta_1)\sin(\zeta_1) \end{bmatrix} \tag{90}$$

Given fixed values for (η_1,ζ_1), and an initial value for r_1, we can use equations 87-90 to evaluate r_2 and η_2, and use equation 12 to evaluate the new value of r_1. This gives an iteration for r_1.

If we use η_1 and ζ_1 as coordinates on the two-dimensional intercept surface, we can use the iteration defined above to solve for the corresponding values of r_1. Then equation 8 gives the Cartesian coordinates of the point.

When $\eta_1 = \pi$, equation 12 is satisfied by $r_1 = 0$. This point is on the intercept surface because the evading aircraft could fly on any circle just large enough so that the aircraft returns to its starting point in the time it takes the missile to get to the aircraft's starting point.

In the iterative procedure, we will make use of the following formula for η_2 in terms of los and γ:

$$
\begin{aligned}
r_2 \cos(\eta_2) &= \frac{\underline{V}_2(0)^T}{\|\underline{V}_2(0)\|} [\underline{r} - \underline{r}_2(0)] \\
&= \frac{\underline{V}_2(0)^T}{\|\underline{V}_2(0)\|} [\underline{r}_1(0) - \underline{r}_2(0)] + \frac{\underline{V}_2(0)^T}{\|\underline{V}_2(0)\|} [\underline{r} - \underline{r}_1(0)] \\
&= r_{12} \cos(\text{los}) + r_1 \cos(\gamma)
\end{aligned}
\tag{91}
$$

To determine a two-dimensional array of points $r_1(\eta_{1_j}, \zeta_{1_i})$ on the intercept surface and evaluate the cost functional at each point, we can use the following procedure (given in more detail later):

k0 = (v1/v2)*sin(eta1)/eta1

cos_los = - x2_minus_x1_unit dot V2_unit

Set $r_1(\eta_{1_{-1}},\zeta_{1_i}) = 0 = r_1(\eta_{1_{-2}},\zeta_{1_i})$

For j = 0 to 30

$\eta_{1_j} = ((30-j)/30)(\pi) + \varepsilon$ $\varepsilon>0$ to avoid (sin(0))/0

se1 = sin(eta1) ce1 = cos(eta1)

For i = -30 to 30

$\zeta_{1_i} = (i/30)\pi$

sz1 = sin(zeta1) cz1 = cos(zeta1)

x_minus_x1_unit = [V1_unit, V1_perp]*[ce1, se1*cz1, se1*sz1]

cos_phi = x2_minus_x1_unit dot x_minus_x1_unit

cos_gam = V2_unit dot x_minus_x1_unit

Initialize $r_1 = 2r_1(\eta_{1_{j-1}},\zeta_{1_i}) - r_1(\eta_{1_{j-2}},\zeta_{1_i})$ (linear extrapolation)

Do four iterations of: (equations 12 and 87-91)

r1_over_r2_old = r1_over_r2

r2_over_r12 = sqrt(r1_over_r12^2 -2*cos_phi*r1_over_r12 + 1)

eta2 = acos((cos_los+r1_over_r12*cos_gam)/r2_over_r12)

k_of_eta = k0 * eta2 / sin(eta2)

r1_over_r12 = k_of_eta * r2_over_r12

r1(i,j) = r1_over_r12*r12

r2(i,j) = r2_over_r12*r12

T1 = (r1/v1) * eta1 / sin(eta1) save (i,j) if max time

T2 = (r2/v2) * eta2 / sin(eta2) T1 = T2 if converged

accel1(i,j) = 2 * eta1 * v1 / T1 (angle*radius=v*T accel=v*v/r)

accel2(i,j) = 2 * eta2 * v2 / T2 (angle*radius=v*T accel=v*v/r)

End i loop

End j loop

$$\underline{accel}_i = [\underline{1}_{r_i}, \underline{h}_i] \begin{bmatrix} \sin(\eta_i) \\ \cos(\eta_i) \end{bmatrix} \frac{2\eta_i \|\underline{V}_i(0)\|}{T_i} \qquad i = 1, 2 \tag{92}$$

so

$$\|\underline{accel}_i\| = \frac{2\|V_i(0)\|^2 \sin(\eta_i)}{r_i} \qquad i = 1, 2 \tag{93}$$

If we assume 10 floating-point operations each for computing atan2, sin, and sqrt, then evaluation of the inner loop equations takes approximately 31 floating-point operations. Doing 4 iterations on a 30×60 grid requires a 31*4*30*60 = 223,000 floating-point operations. The outer loops of the iteration account for an additional 27,000 floating-point operations. This gives a total of approximately 250,000 floating-point operations. A 1 Mflop computer could update this guidance algorithm at 4 Hz.

Convergence of Iteration

Theorem 3: The iteration converges at points on the solution surface where:

$$(\underline{r} - \underline{r}_1(0))^T \frac{dF(\underline{r})}{d\underline{r}} < 0 \tag{94}$$

i.e., whenever the surface is star-shaped.

Proof:

The iteration is of the form:

$$r_{1_{new}} = G(r_{1_{old}}, \underline{1}_{r_1}) \tag{95}$$

where $\underline{1}_{r_1}$ is a unit vector based at the point $\underline{r}_1(0)$, and r_1 is the distance along that unit vector. For each fixed value of the unit vector, the iteration has a fixed point at the value of r_1 that gives the point on the solution surface:

$$\underline{r} = \underline{r}_1(0) + r_1 \underline{1}_{r_1} \tag{96}$$

We define the function G by:

$$G = r_1 - F(\underline{r}) = k\, r_2 \tag{97}$$

Then

$$\frac{dG}{dr_1} = 1 - \frac{dF}{dr_1} =$$

$$1 - \underline{1}_{r_1}{}^T \frac{dF}{d\underline{r}} =$$

$$1 - \left[1 - k\, \underline{1}_{r_1}{}^T[\underline{1}_{r_2}, \underline{h}_2] \left[\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)}\right]\right] \tag{98}$$

$$= k\, \underline{1}_{r_1}{}^T[\underline{1}_{r_2}, \underline{h}_2] \left[\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)}\right]$$

The iteration converges when $|\frac{dG}{dr_1}| < 1$, which is ensured when

$$k \sqrt{1 + (\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)})^2} < 1 \tag{99}$$

or

$$\frac{\|\underline{V}_2(0)\|}{\|\underline{V}_1(0)\|} \geq \frac{\eta_2}{\sin(\eta_2)} \sqrt{1 + \left[\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)}\right]^2} \tag{100}$$

Note that this is the same bound found for ensuring that the intercept surface had no singularities in regions of space (values of η_2) that satisfied the above inequality for some given $\frac{\|\underline{V}_2(0)\|}{\|\underline{V}_1(0)\|}$.

Finally,

$$\underline{1}_{r_1}^T \frac{dF}{d\underline{r}} = 1 - \frac{dG}{dr_1} > 0 \qquad \text{whenever} \qquad \left|\frac{dG}{dr_1}\right| < 1 \qquad (101)$$

So the iteration converges whenever $\underline{1}_{r_1}^T \frac{dF}{d\underline{r}} > 0$

Note that the function $\frac{1}{\eta_2} - \frac{1}{\tan(\eta_2)}$ has value 0, and slope +1 at $\eta_2 = 0$, and is monotonically increasing, going to infinity at $\eta_2 = \pi$.

The function $k(\eta_2)$ is monotonically increasing as η_2 goes from 0 to π. The function $k(\eta_2)$ goes to $+\infty$ as η_2 goes to π.

5. Algebraic Algorithm for Computing the Intercept Surface

For cases where the iteration defined in the previous section does not converge, we can solve algebraically for all points on the two-dimensional intercept surface in $\mathbf{R}^3$. We can use η_1 and η_2 as coordinates on the two-dimensional surface. For fixed η_1 and η_2, equation 12 defines an ellipsoid in $\mathbf{R}^3$. Equation 8 indicates that fixing η_1 defines a circular cone in $\mathbf{R}^3$ with vertex at $\underline{r}_1(0)$ and axis-of-symmetry $\underline{V}_1(0)$. Equation 8 also indicates that fixing η_2 defines a circular cone in $\mathbf{R}^3$ with vertex at $\underline{r}_2(0)$ and axis-of-symmetry $\underline{V}_2(0)$. The intersection of these three quadratic surfaces in $\mathbf{R}^3$ results in 8 (some possibly complex) solution points for each fixed pair (η_1,η_2).

To get three second-order algebraic equations, we will use Cartesian coordinates for the ellipsoid in equation 12, and for the two cone equations: η_1 = constant, η_2 = constant. Squaring equation 12 gives:

$$\|\underline{r} - \underline{r}_1(0)\|^2 = \left[\frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|}\frac{\eta_2}{\eta_1}\frac{\sin(\eta_1)}{\sin(\eta_2)}\right]^2 \|\underline{r} - \underline{r}_2(0)\|^2 \tag{102}$$

If we let:

$$k = \frac{\|\underline{V}_1(0)\|}{\|\underline{V}_2(0)\|}\frac{\eta_2}{\eta_1}\frac{\sin(\eta_1)}{\sin(\eta_2)} \tag{103}$$

then equation 102 becomes:

$$\|\underline{r} - \underline{r}_1(0)\|^2 = k^2 \|\underline{r} - \underline{r}_2(0)\|^2 \tag{104}$$

which is a second-order algebraic equation for a circular ellipsoid in $\mathbf{R}^3$.

Setting η_i = constant gives a circular half-cone with vertex at $\underline{r}_i(0)$ and axis-of-symmetry $\underline{V}_i(0)$

$$\left[\underline{r} - \underline{r}_i(0)\right]^T \underline{V}_i(0) = \cos(\eta_i)\, \|\underline{V}_i(0)\|\, \|\underline{r} - \underline{r}_i(0)\| \qquad i=1,2 \tag{105}$$

Squaring equation 105 gives a second-order polynomial equation:

$$0 = \left[\underline{r} - \underline{r}_i(0)\right]^T \left[\underline{V}_i(0)\underline{V}_i(0)^T - I\cos^2(\eta_i)\, \|\underline{V}_i(0)\|^2\right]\left[\underline{r} - \underline{r}_i(0)\right] \qquad i=1,2 \tag{106}$$

Equation 106 is for a full cone, instead of a half-cone, since the sign of $\cos(\eta_i)$ is ignored.

Together, equations 104 and 106 are three second-order equations in the three Cartesian components of $\underline{r}$. By Bezout's theorem, we expect up to eight solutions for each fixed set of (η_1,η_2).

To solve the three equations, we first rotate, translate, and uniformly stretch the coordinates to move the two cone vertices to $x = \pm 1$:

$$\underline{r}_1(0) \rightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \underline{r}_2(0) \rightarrow \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} \tag{107}$$

We then rotate the coordinate system about the x axis (leaving the above two points fixed) till $\underline{V}_1(0)$ has no z component.

In this new coordinate system, let:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \underline{r} \qquad \begin{bmatrix} a \\ b \\ 0 \end{bmatrix} = \frac{\underline{V}_1(0)}{\cos(\eta_1)\|\underline{V}_1(0)\|} \qquad \begin{bmatrix} d \\ e \\ f \end{bmatrix} = \frac{\underline{V}_2(0)}{\cos(\eta_2)\|\underline{V}_2(0)\|} \tag{108}$$

The three second-order equations can then be written as:

$$0 = \begin{bmatrix} x-1 \\ x+1 \\ y \\ z \end{bmatrix}^T \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & k^2 & 0 & 0 \\ 0 & 0 & k^2-1 & 0 \\ 0 & 0 & 0 & k^2-1 \end{bmatrix} \begin{bmatrix} x-1 \\ x+1 \\ y \\ z \end{bmatrix} \qquad \text{ellipsoid} \tag{109}$$

$$0 = \begin{bmatrix} x-1 \\ x+1 \\ y \\ z \end{bmatrix}^T \begin{bmatrix} a^2-1 & 0 & a\,b & 0 \\ 0 & 0 & 0 & 0 \\ a\,b & 0 & b^2-1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x-1 \\ x+1 \\ y \\ z \end{bmatrix} \qquad \text{cone}_1 \tag{110}$$

$$0 = \begin{bmatrix} x-1 \\ x+1 \\ y \\ z \end{bmatrix}^T \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & d^2-1 & d\,e & d\,f \\ 0 & d\,e & e^2-1 & e\,f \\ 0 & d\,f & e\,f & f^2-1 \end{bmatrix} \begin{bmatrix} x-1 \\ x+1 \\ y \\ z \end{bmatrix} \qquad \text{cone}_2 \tag{111}$$

To continue the solution process, we introduce a bi-rational transformation:

$$\begin{bmatrix} 1 \\ u \\ v \\ w \end{bmatrix} = \frac{1}{x-1} \begin{bmatrix} x-1 \\ x+1 \\ y \\ z \end{bmatrix} \qquad \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \frac{1}{u-1} \begin{bmatrix} u+1 \\ 2v \\ 2w \end{bmatrix} \tag{112}$$

Since this only differs by a scalar from the old 4-vector, the three matrices in equations 109, 110, and 111 remain unchanged:

$$0 = \begin{bmatrix} 1 \\ u \\ v \\ w \end{bmatrix}^T \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & k^2 & 0 & 0 \\ 0 & 0 & k^2-1 & 0 \\ 0 & 0 & 0 & k^2-1 \end{bmatrix} \begin{bmatrix} 1 \\ u \\ v \\ w \end{bmatrix} \qquad \text{ellipsoid} \tag{113}$$

$$0 = \begin{bmatrix} 1 \\ u \\ v \\ w \end{bmatrix}^T \begin{bmatrix} a^2 - 1 & 0 & a\,b & 0 \\ 0 & 0 & 0 & 0 \\ a\,b & 0 & b^2 - 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ u \\ v \\ w \end{bmatrix} \qquad \text{cone}_1 \tag{114}$$

$$0 = \begin{bmatrix} 1 \\ u \\ v \\ w \end{bmatrix}^T \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & d^2 - 1 & d\,e & d\,f \\ 0 & d\,e & e^2 - 1 & e\,f \\ 0 & d\,f & e\,f & f^2 - 1 \end{bmatrix} \begin{bmatrix} 1 \\ u \\ v \\ w \end{bmatrix} \qquad \text{cone}_2 \tag{115}$$

The next step in the solution is to get a combination of equations 113 and 114 to be of the form: $u^2 = f(v)$ and $w^2 = g(v)$ where f(v) and g(v) are quadratic polynomials in v. Then equation 115 will be both rearranged and squared two times to get a fourth-order equation in (u^2, v^2, w^2). By replacing u^2 and w^2 with f(v) and g(v), we then get an eighth-order polynomial in v alone. The eight v solutions can be used to obtain the eight corresponding values of u and w.

Taking equation 113 plus k^2–1 times equation 114 gives an equation of the form $u^2 = f(v)$:

$$k^2\,u^2 = 1 - (k^2 - 1)\,[(a^2 - 1) + 2\,a\,b\,v + b^2\,v^2] \tag{116}$$

Equation 114 is already of the form $w^2 = g(v)$:

$$w^2 = (a^2 - 1) + 2\,a\,b\,v + (b^2 - 1)\,v^2 \tag{117}$$

We are now ready to begin working on getting equation 115 in the form of a fourth-order equation in (u^2, v^2, w^2). We begin by using M to denote the 3×3 nonzero submatrix in equation 115.

$$0 = \begin{bmatrix} u \\ v \\ w \end{bmatrix}^T \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{12} & m_{22} & m_{23} \\ m_{13} & m_{23} & m_{33} \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{118}$$

Rearranging equation 118 gives:

$$\begin{bmatrix} u \\ v \end{bmatrix}^T \begin{bmatrix} m_{11} & m_{12} \\ m_{12} & m_{22} \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} + w\, m_{33}\, w = -2\, w\, [m_{13},\, m_{23}] \begin{bmatrix} u \\ v \end{bmatrix} \tag{119}$$

Note that when the $[\underline{r}_1(0),\underline{r}_2(0),\underline{V}_1(0),\underline{V}_2(0)]$ data is planer, we get f=0, so $m_{13} = 0 = m_{23}$. In that case, we can separate the uv dependence in equation 119, replace u^2 and w^2 with quadratic polynomials in v, then square the resulting equation to get a quartic equation in v.

For the non-planar case, we can square both sides of equation 119 to get:

$$\left\{ \begin{bmatrix} u \\ v \end{bmatrix}^T \begin{bmatrix} m_{11} & m_{12} \\ m_{12} & m_{22} \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} + m_{33}\, w^2 \right\}^2 = 4\, w^2 \begin{bmatrix} u \\ v \end{bmatrix} \left\{ \begin{bmatrix} m_{13} \\ m_{23} \end{bmatrix} [m_{13},\, m_{23}] \right\} \begin{bmatrix} u \\ v \end{bmatrix} \tag{120}$$

Expanding and rearranging equation 120 gives:

$$\begin{bmatrix} u^2 \\ v^2 \\ w^2 \end{bmatrix}^T \begin{bmatrix} m_{11}^2 & m_{11}m_{22}+2m_{12}^2 & m_{33}m_{11}-2m_{13}^2 \\ & m_{22}^2 & m_{33}m_{22}-2m_{23}^2 \\ \text{symmetric} & & m_{33}^2 \end{bmatrix} \begin{bmatrix} u^2 \\ v^2 \\ w^2 \end{bmatrix} \tag{121}$$

$$= 4\, u\, v\, [-m_{12}m_{11}\, u^2 - m_{12}m_{22}\, v^2 + (2m_{13}m_{23} - m_{33}m_{12})w^2]$$

Squaring both sides of equation 121 gives:

$$0 = \begin{bmatrix} u^4 \\ v^4 \\ w^4 \end{bmatrix}^T A \begin{bmatrix} u^4 \\ v^4 \\ w^4 \end{bmatrix} + 4 \begin{bmatrix} u^4 \\ v^4 \\ w^4 \end{bmatrix}^T B \begin{bmatrix} v^2\, w^2 \\ u^2\, w^2 \\ u^2\, v^2 \end{bmatrix} \tag{122}$$

where the entries of A are given by:

$$\begin{aligned} a_{ii} &= m_{ii}^4 \qquad i = 1,2,3 \\ a_{ij} &= 3m_{ii}^2 m_{jj}^2 + 8m_{ij}^2(m_{ij}^2 - m_{ii}m_{jj}) \end{aligned} \tag{123}$$

and the entries of B are given by:

$$\begin{aligned} b_{ii} &= m_{ii}^2(3m_{jj}m_{kk} - 2m_{jk}^2) - 4m_{ii}(m_{jj}m_{ik}^2 + m_{kk}m_{ij}^2) + 8m_{ik}m_{ij}(2m_{ii}m_{jk} - m_{ik}m_{ij}) \\ b_{ij} &= m_{ii}^2(m_{ii}m_{kk} - 2m_{ik}^2) \qquad (i,j,k) \text{ cyclic permutation of } (1,2,3) \end{aligned} \tag{124}$$

If we substitute equations 116 and 117 into equation 122 for each occurrence of powers of u^2 and w^2, we get an eighth-order polynomial in v alone. We can put the coefficients of this polynomial into an 8×8 companion matrix, whose eigenvalues will be the roots of the polynomial.

If we substitute equations 116 and 117 into equation 121 for each occurrence of u^2 and w^2, we get a linear equation in u, whose coefficients are polynomials in v. By evaluating these coefficients with the 8 values of v found above, we get the 8 corresponding values of u.

If we substitute equations 116 and 117 into equation 119 for each occurrence of u^2 and w^2, we get a linear equation in w whose coefficients are polynomials in u and v. By evaluating these coefficients with the 8 values of u and v found above, we get the 8 corresponding values of w.

Finally, we can use the right part of equation 112 to convert [u,v,w] back to [x,y,z]. This gives us the [x,y,z] values on the two-dimensional intercept surface for each value of the (η_1, η_2) coordinates.

A numerical example is given below.

Input Data:

r1_ = [0.5163; 0.3190; 0.9866]

r2_ = [0.0606; 0.9047; 0.5045]

V1_ = [0.2363; 0.0490; -0.1546]

V2_ = [0.0782; -0.3340; 0.3554]

eta1 = 1.5407

eta2 = 0.5354

Computed Results:

k = 0.5802

los = 0.4000

norm(V1_)/norm(V2_) = 0.5802

eta2c = (pi - asin(norm(V1_)/norm(V2_)) + los)/2 = 1.4613

(norm(V2_)/norm(V1_))*(sin(eta2c)/eta2c)*sin(eta2c-los) = 1.0236

def = V2/(norm(V2)*cos(eta2)) = [1.0709; -0.4527; 0.0071]

ab0 = V1/(norm(V1)*cos(eta1)) = [0.5895; 33.2048; 0]

order8_poly = 1936656295. 290948514. -27327350. -4381463.6 173893.7
22468.886 -702.23409 -39.84748 1.34520

uv0_coeff = 55649.908255 3712.345422 -424.606279 -17.178161 1.15982968

uv1_coeff = 440.828064411611 15.485303259795 -2.088513070591

[w0_coeff, w1_coeff] = [0.12675655356533, 0.07321379932120]

uvw =

8.3079	-0.1243	3.4003
6.3804	-0.0991	-2.5165
-7.3963	0.0773	-2.9874
-5.8503	0.0571	2.2673
-4.9642	-0.0805	-1.8345
-4.4625	-0.0738	1.5786
5.3506	0.0505	2.0264
4.7569	0.0426	-1.7313

% Bi-Rational transformation

x(j) = (u(j)+1)/(u(j)-1);

y(j) = 2*v(j)/(u(j)-1);

z(j) = 2*w(j)/(u(j)-1);

xyz =

1.2737	-0.0340	0.9306
1.3717	-0.0368	-0.9354
0.7618	-0.0184	0.7116
0.7080	-0.0167	-0.6619
0.6647	0.0270	0.6152
0.6339	0.0270	-0.5780
1.4597	0.0232	0.9315
1.5323	0.0227	-0.9217

These eight solutions also satisfy the HALF-cone restrictions of equations 105.

The computational cost of evaluating equations 103, 108, 118, 123, and 124 is approximately 240 floating-point operations. The cost of computing the eigenvalues of an 8×8 companion matrix is approximately $4 * 8^3 = 2048$ floating-point operations. The cost of back-substitution into equations 121, 119, and 112 is approximately 70 floating-point operations. This gives a total of approximately 2400 floating-point operations for each fixed set of (η_1, η_2). If we use a 30×30 grid for (η_1, η_2), then the total computational cost is approximately 2.1 million floating-point operations. This is approximately eight times as expensive as the iterative solution. A 2 Mflop computer could update the algebraic solution of the guidance algorithm at approximately 1 Hz.

Degeneracy in Back Substitution:

When equation 122 (the eighth-order polynomial in v) is the square of a fourth-order polynomial, equation 121 (made linear in u) gives a single u for each of the four v roots. However, back-substitution equation 119 (made linear in w) reduces to $0 = 0$, in which case we can use equation 117 which gives two w solutions for each of the four separate (u,v) pairs.

When equation 122 (the eighth-order polynomial in v) is the fourth power of a second-order polynomial, back-substitution equations 119 (made linear in w) and 121 (made linear in u) both reduce to $0 = 0$, in which case we can use equations 116 and 117 which give four (u,w) solutions for each of the two separate v roots.

6. Examples

Figures 10 through 14 at the end of this section are from an example where the missile's acceleration constraint is not exceeded anywhere on the intercept surface (except near the end-game). Since the evading aircraft is slower than the pursuing missile, and the missile is not on an acceleration constraint, the aircraft is completely enclosed by the intercept surface and cannot escape. Therefore the aircraft heads for the point on the intercept surface that results in the longest intercept time (in hopes that the missile's fuel runs out). The missile assumes that the aircraft has headed for the point with the longest intercept time, so the missile also heads for that point.

The guidance algorithm itself could update at 80 Hz on the 120 MHz Pentium that it was running on, but drawing the smoothly-lit color intercept surface on the screen each time took .5 seconds, so in this example, the intercept surface was only recomputed every .5 seconds. Each .5 second, both vehicles get to pick new values for their optimal constant normal acceleration inputs. In this example, it takes 38 seconds for the missile and aircraft to intercept.

Figures 15 though 18 are from the second example. In the second example, the same initial positions and velocities were used as in the first example, but the aircraft makes no attempt to maneuver. The aircraft flies on a straight path, while the missile still assumes that it will head for the point on the intercept surface corresponding to the longest intercept time. The intercept surface shrinks to a point as time progresses, causing the point the missile is headed for and the point the aircraft is headed for, to come together. Since the aircraft has not headed for the point that would result in the longest time to intercept, intercept occurs sooner than in the first example (29 seconds instead of 38 seconds).

Both of the above examples were done using the iterative solution for the intercept surface.

Near the lower-right corner of figures 10 through 18 is printed an error. e.g. error = .0013 in figure 10. This is the amount that r_1 changed on the fourth (final) iteration. The vehicles start out around 8 "units of length" apart in each of the two examples, and the error is measured in those same units.

Example 1 (5 frames from a simulation, update rate = .5 seconds)

Aircraft on piece-wise circular path

Missile on piece-wise circular path

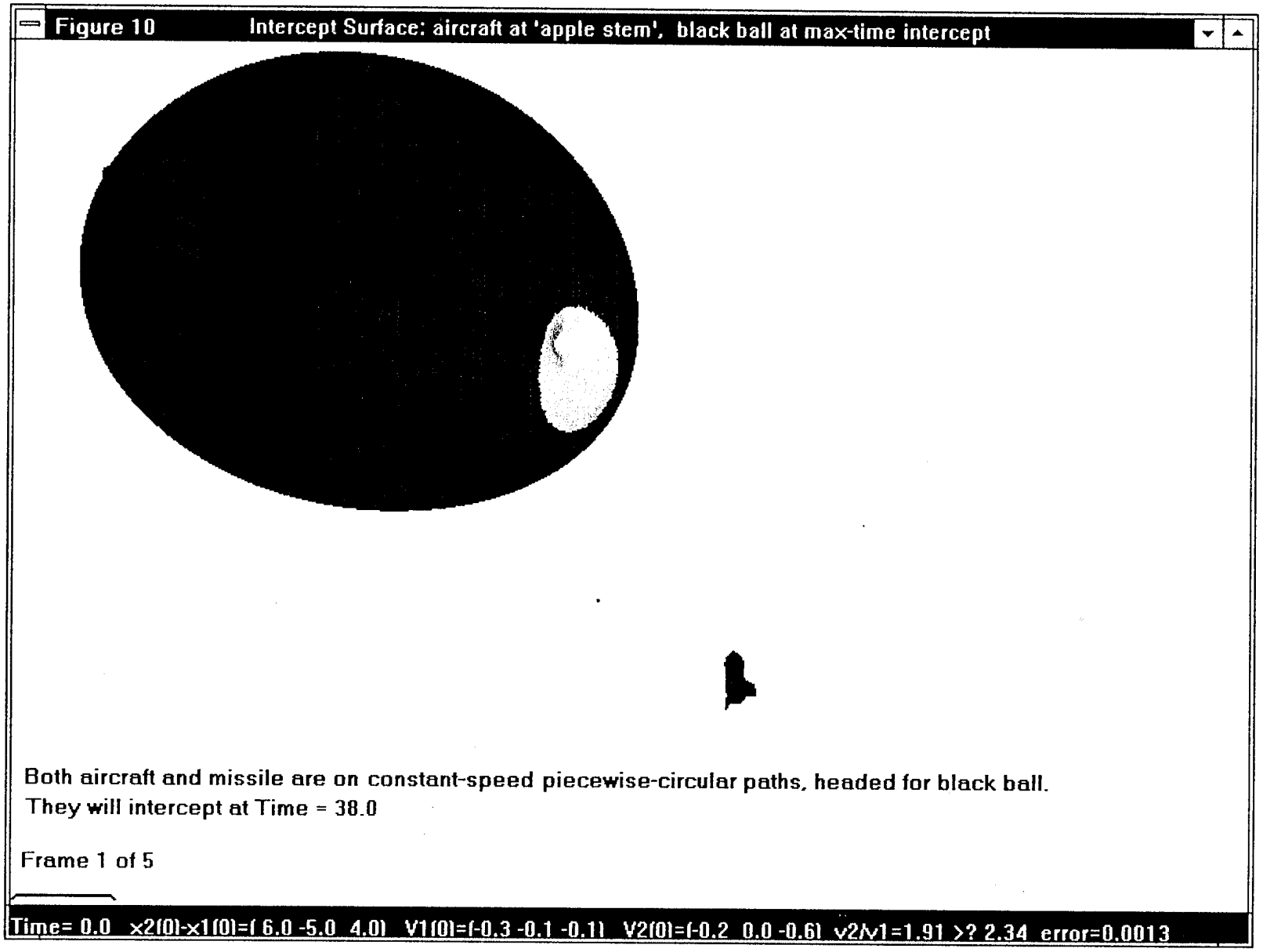
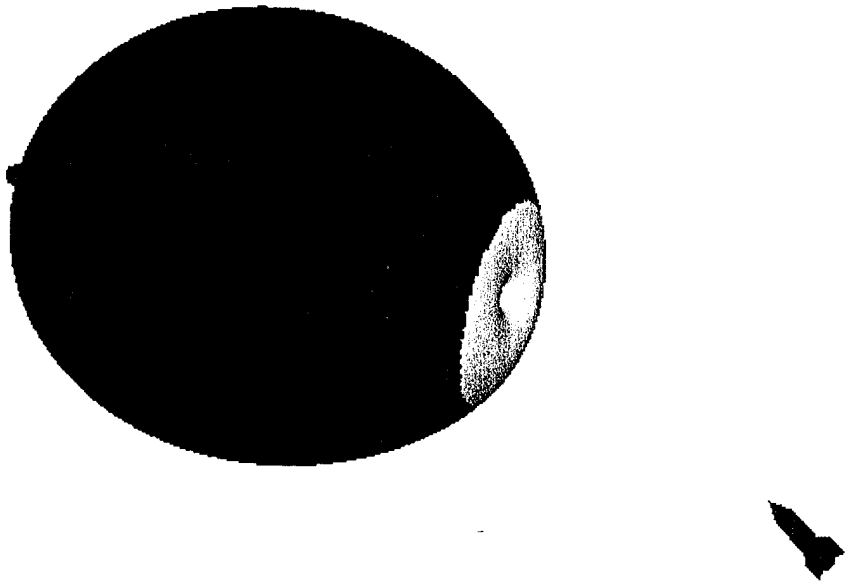


Figure 10
Intercept Surface: aircraft at 'apple stem', black ball at max-time intercept
Both aircraft and missile are on constant-speed piecewise-circular paths, headed for black ball.
They will intercept at Time = 38.0
Frame 1 of 5
Time= 0.0 x2(0)-x1(0)=(6.0 -5.0 4.0) V1(0)=(-0.3 -0.1 -0.1) V2(0)=(-0.2 0.0 -0.6) v2/v1=1.91 >? 2.34 error=0.0013

Figure 11 Intercept Surface: aircraft at 'apple stem', black ball at max-time intercept



Both aircraft and missile are on constant-speed piecewise-circular paths, headed for black ball.
They will intercept at Time = 38.0

Frame 2 of 5

Time=10.0 x2(0)-x1(0)=[6.0 -5.0 4.0] V1(0)=[-0.3 -0.1 -0.1] V2(0)=[-0.2 0.0 -0.6] v2/v1=1.91 >? 2.13 error=0.00073

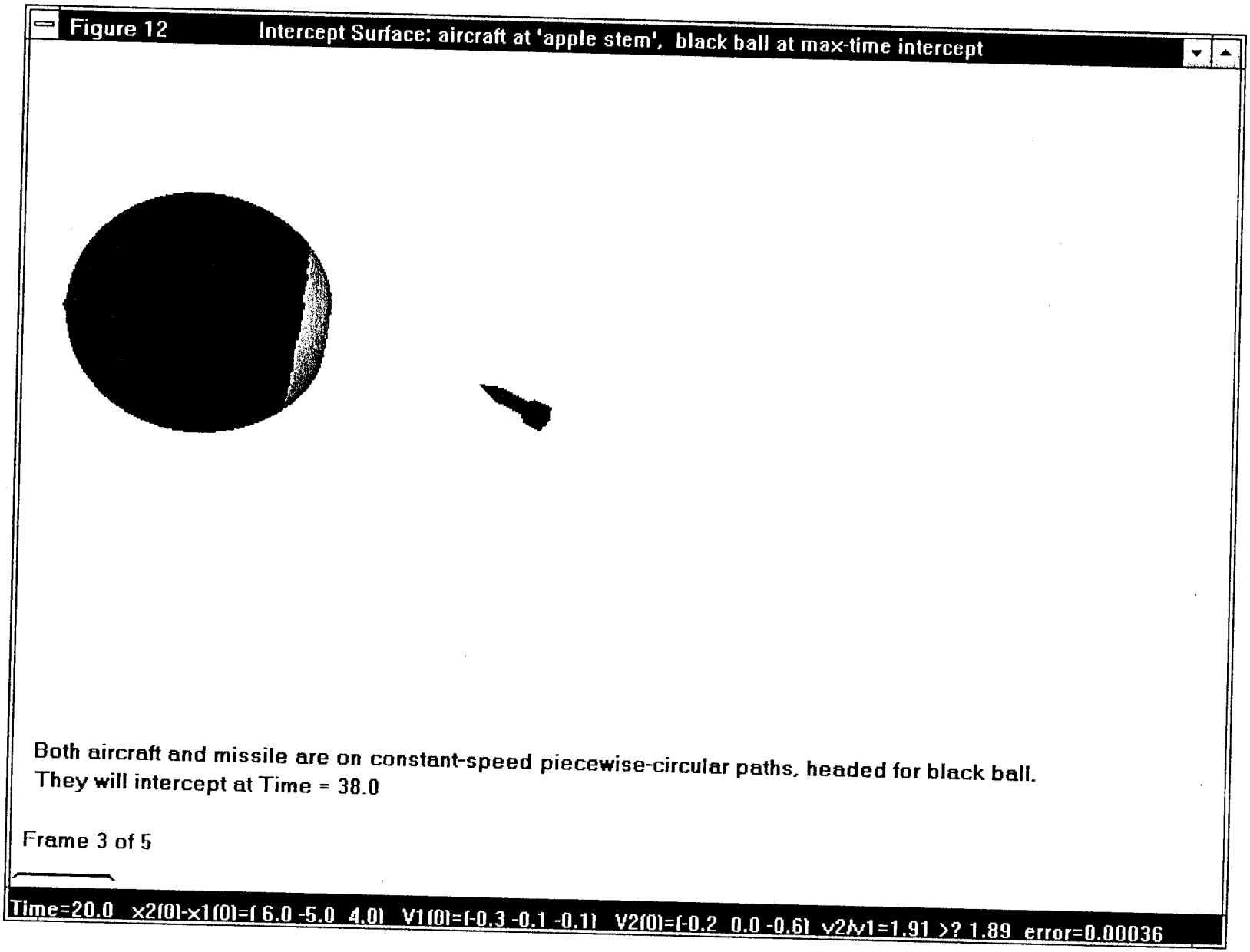


Figure 12
Intercept Surface: aircraft at 'apple stem', black ball at max-time intercept
Both aircraft and missile are on constant-speed piecewise-circular paths, headed for black ball.
They will intercept at Time = 38.0
Frame 3 of 5
Time=20.0 x2(0)-x1(0)=(6.0 -5.0 4.0) V1(0)=(-0.3 -0.1 -0.1) V2(0)=(-0.2 0.0 -0.6) v2/v1=1.91 >? 1.89 error=0.00036

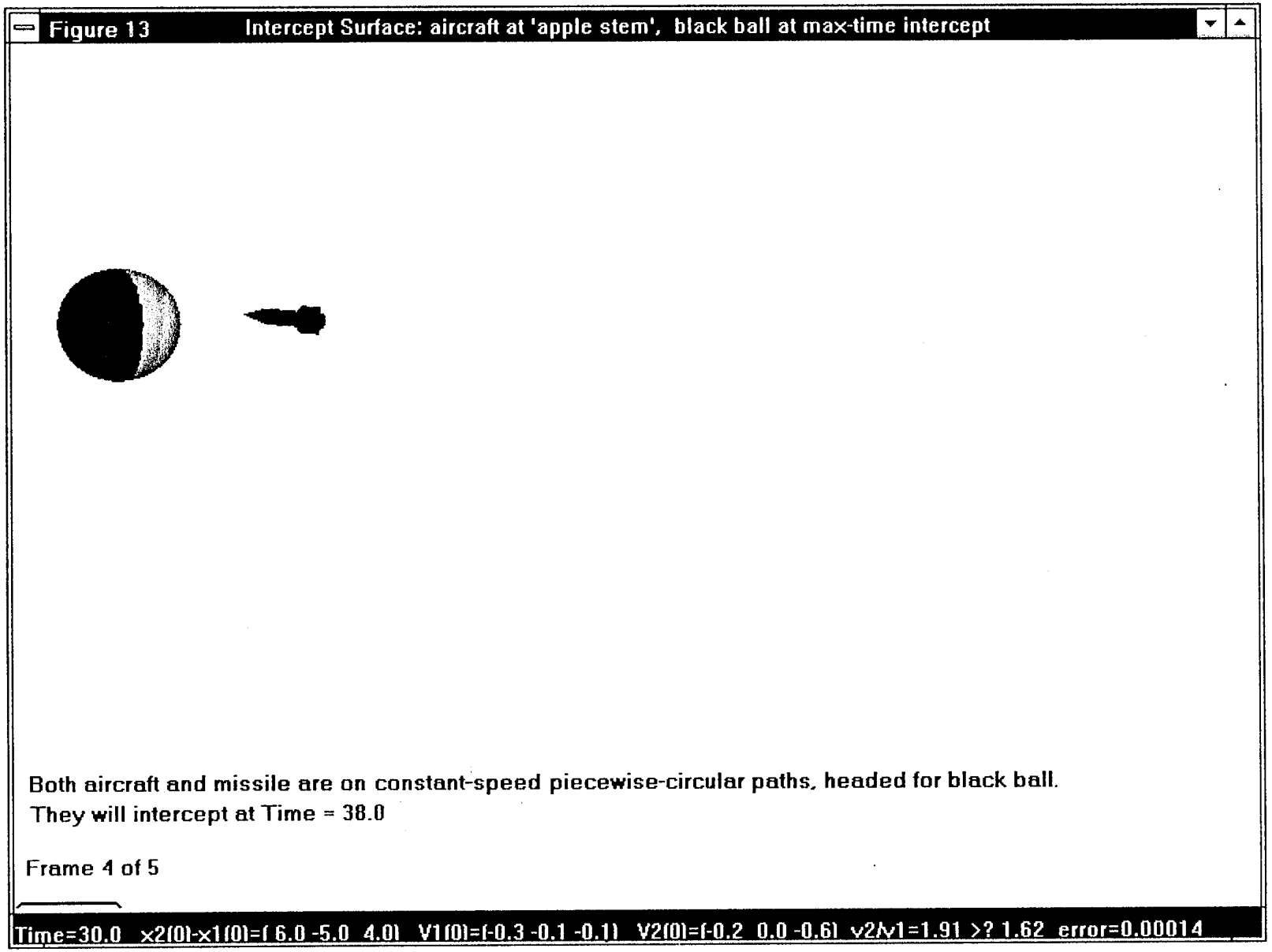


Figure 13
Intercept Surface: aircraft at 'apple stem', black ball at max-time intercept
Both aircraft and missile are on constant-speed piecewise-circular paths, headed for black ball.
They will intercept at Time = 38.0
Frame 4 of 5
Time=30.0 x2(0)-x1(0)=(6.0 -5.0 4.0) V1(0)=(-0.3 -0.1 -0.1) V2(0)=(-0.2 0.0 -0.6) v2/v1=1.91 >? 1.62 error=0.00014

Figure 14 Intercept Surface: aircraft at 'apple stem', black ball at max-time intercept

Both aircraft and missile are on constant-speed piecewise-circular paths, headed for black ball.
They will intercept at Time = 38.0

Frame 5 of 5

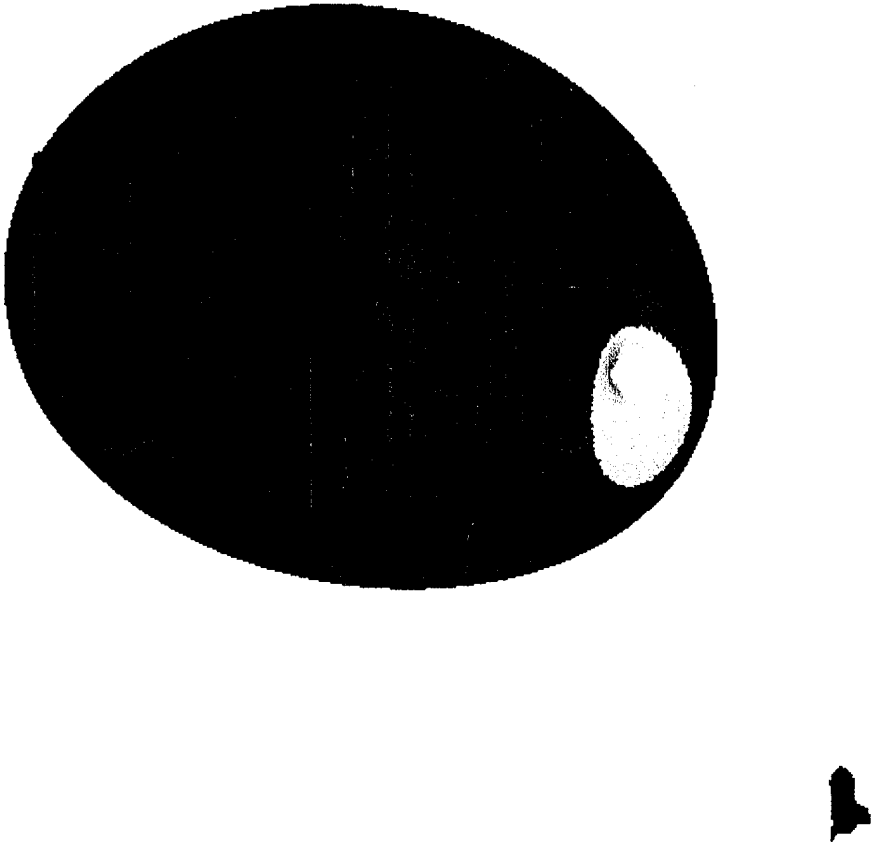
Time=37.0 x2(0)-x1(0)=(6.0 -5.0 4.0) V1(0)=(-0.3 -0.1 -0.1) V2(0)=(-0.2 0.0 -0.6) v2/v1=1.91 >? 1.39 error=1.1e-005

Example 2 (4 frames from a simulation, update rate = .5 seconds)

Aircraft on straight line path

Missile on piece-wise circular path

Figure 15 Intercept Surface: aircraft at 'apple stem', black ball at max-time intercept



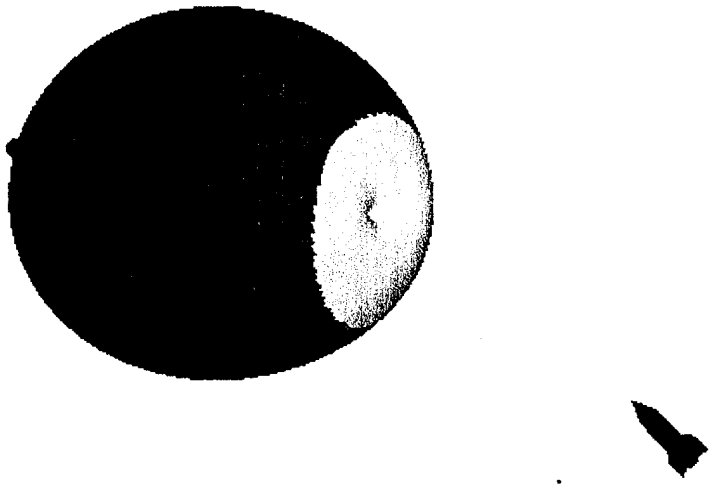
The aircraft is on a constant-speed straight-line path.

The missile is on a constant-speed piecewise-circular path, headed for the black ball.
They will intercept at Time = 29.0

Frame 1 of 4

Time= 0.0 x2(0)-x1(0)=(6.0 -5.0 4.0) V1(0)=(-0.3 -0.1 -0.1) V2(0)=(-0.2 0.0 -0.6) v2/v1=1.91 >? 2.34 error=0.0013

Figure 16 Intercept Surface: aircraft at 'apple stem', black ball at max-time intercept



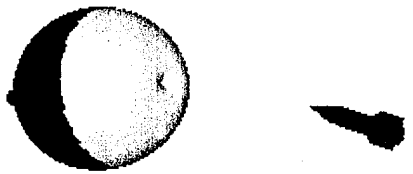
The aircraft is on a constant-speed straight-line path.

The missile is on a constant-speed piecewise-circular path, headed for the black ball.
They will intercept at Time = 29.0

Frame 2 of 4

Time=10.0 x2(0)-x1(0)=(6.0 -5.0 4.0) V1(0)=(-0.3 -0.1 -0.1) V2(0)=(-0.2 0.0 -0.6) v2/v1=1.91 >? 2.04 error=0.00043

Figure 17 Intercept Surface: aircraft at 'apple stem', black ball at max-time intercept



The aircraft is on a constant-speed straight-line path.

The missile is on a constant-speed piecewise-circular path, headed for the black ball.
They will intercept at Time = 29.0

Frame 3 of 4

Time=20.0 x2(0)-x1(0)=(6.0 -5.0 4.0) V1(0)=(-0.3 -0.1 -0.1) V2(0)=(-0.2 0.0 -0.6) v2/v1=1.91 >? 1.63 error=0.00016

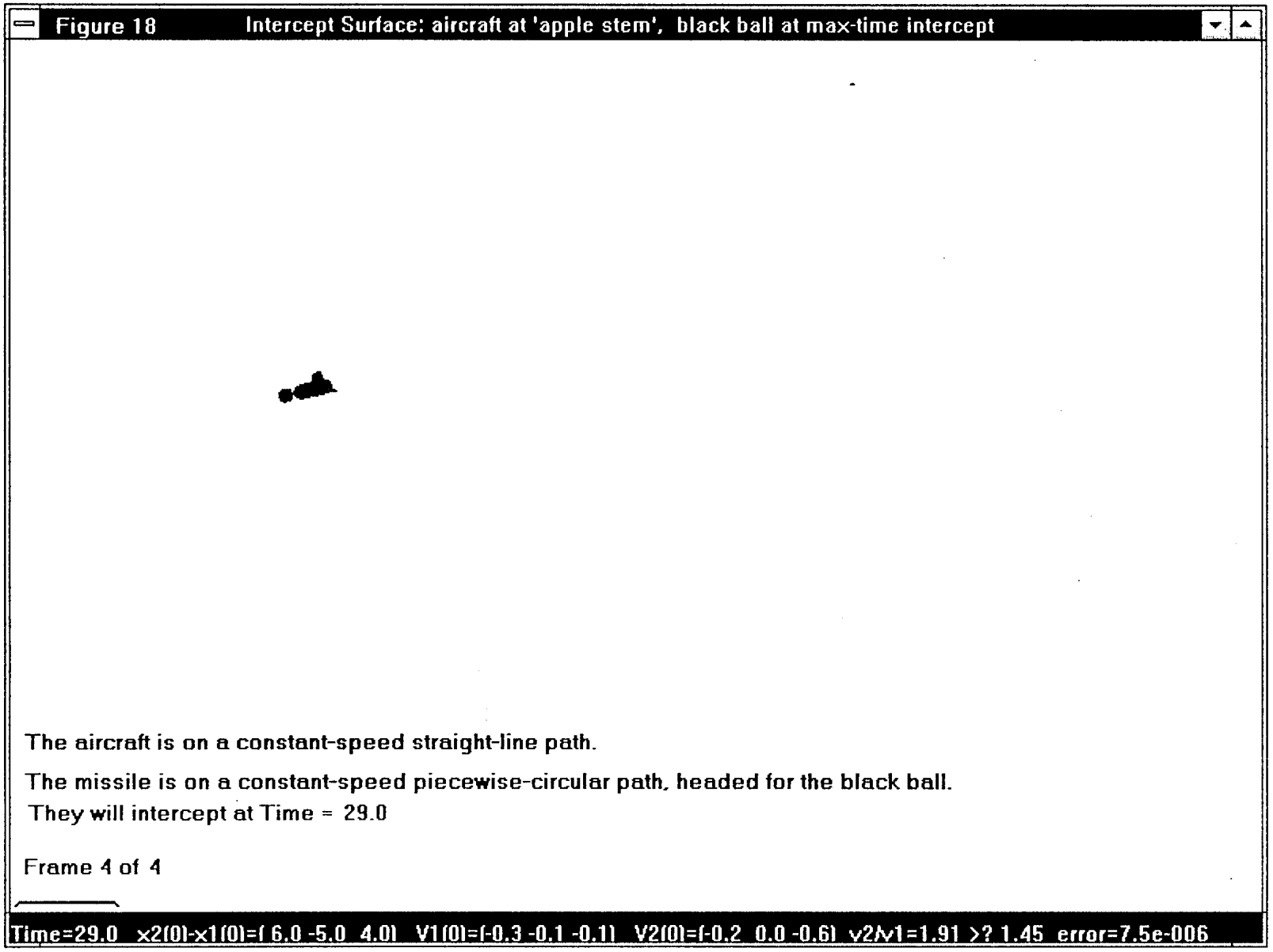


Figure 18 Intercept Surface: aircraft at 'apple stem', black ball at max-time intercept
The aircraft is on a constant-speed straight-line path.
The missile is on a constant-speed piecewise-circular path, headed for the black ball.
They will intercept at Time = 29.0
Frame 4 of 4
Time=29.0 x2(0)-x1(0)=[6.0 -5.0 4.0] V1(0)=[-0.3 -0.1 -0.1] V2(0)=[-0.2 0.0 -0.6] v2/v1=1.91 >? 1.45 error=7.5e-006

7. Summary

The proposed mini-max pursuit-evader algorithm allows both vehicles to maneuver in 3D space and yet is fast enough to run real-time on today's flight computers. It also allows for the possibility of providing the pilot of either a missile-launching aircraft, or a missile-evading aircraft, an intuitively clear display of the current intercept surface with constraints and cost-functional clearly displayed. This information can be used to help the pilot make a missile launch decision.

The drawback of using the algorithm on the missile itself, is that the missile would need velocity and range measurements, either from its own sensors or from the launching aircraft.

We intend to further investigate this guidance algorithm and its behavior for various types of constraints, cost functionals, and initial conditions. We also would like to compare the results to more conventional mini-max pursuit-evader algorithms that do not make the piece-wise constant speed, piece-wise constant acceleration assumptions that we have used.

The bounds of the form: v1/v2 = H(los) for acceleration limits and for convergence of the iteration should be extended to the case were there is more than one arc in the piece-wise circular trajectories.

Bibleography

Title: On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents.

Author: Dubins, L. E.

Source: American Journal of Mathematics, 1957, vol. 79, pp. 497-516

Title: Some aspects of a realistic three-dimensional pursuit-evasion game

Author: Imado, Fumiaki

Corporate Source: Mitsubishi Electric Corp, Hyogo, Jpn

Source: J. of Guidance, Control, and Dynamics v 16 n 2 Mar-Apr 93. p 289-293

Abstract: A three-dimensional pursuit-evasion game between a realistic missile and an aircraft is studied employing point-mass models for both vehicles. Since a direct method to solve this complicated mini-max problem is too time-consuming, the study is conducted by carrying out massive simulations in the parameter space of initial conditions and guidance law parameters. The important role of information in the opponent's acceleration to the game and the effectiveness of the strategy in rotating the line-of-sight vector are shown. It is found that there exist very few cases where the aircraft can avoid the missile, among them typical air-combat maneuvers such as linear acceleration, high-g barrel roll, split-S, and horizontal-S. (Author abstract) 9 Refs.

Title: Game optimal guidance law synthesis for short range missiles.

Author: Green, A.; Shinar, J.; Guelman, M.

Corporate Source: Technion-Israel Inst of Technology, Haifa, Isr

Source: J. of Guidance, Control, and Dynamics v 15 n 1 Jan-Feb 92. p 191-197

Abstract: A horizontal pursuit-evasion game of kind in the atmosphere between a coasting pursuer with a final velocity constraint and a maneuvering evader of constant speed is

considered. For this model, which is suitable to describe short range missile engagements, the adjoint equations can be integrated analytically. This allows us to determine the optimal strategies of the players on the boundary of the capture set, called the barrier, as a function of the current and final values of the state variables. The main effect of the pursuer's final velocity constraint, an important realistic parameter, neglected in previous studies, is a substantial reduction of the capture zone. However, based on this game solution, a feedback guidance law, suitable for real-time implementation, can be synthesized and compared to other guidance laws. The results show that the corresponding capture set is much larger than the firing envelope of a similar missile guided by proportional navigation with the same final velocity constraint. (Author abstract) 9 Refs.

Title: Proceedings of the 4th International Symposium on Differential Games and Applications.

Author: Anon (Ed.)

Conference Title: Proceedings of the 4th International Symposium on Differential Games and Applications

Conference Location: Helsinki, Finl Conference Date: 1990 Aug 9-10

Source: Lecture Notes in Control and Information Sciences v 156 1991. Publ by Springer-Verlag Berlin, Dept ZSW, Berlin 33, Ger. 292p

Abstract: This conference proceedings contains 29 papers. The papers address topics in differential games and applications. The papers discuss two main categories: classical zero-sum differential games on computational questions as well as engineering applications, and economics and management problems. Many of the papers discuss pursuit-evasion games as applied to air combat.

Title: Application of stochastic differential games to medium-range air-to-air missiles.

Author: Yavin, Y.; De Villiers, R.

Corporate Source: Univ of Pretoria, Pretoria, S Afr

Source: Journal of Optimization Theory and Applications v 67 n 2 Nov 1990 p 355-367

Abstract: Stochastic differential game techniques are applied to compare the performance of a medium-range air-to-air missile for three different thrust-mass profiles. The measure of the performance of the missile is the probability that it will reach a lock-on point with a favorable range of guidance and flight parameters during a fixed time interval $[0,t_f]$.

Title: New results in optimal missile avoidance analysis

Author(s): Shinar, J.; Tabak, R.

Author Affiliation: Technion-Israel Inst. of Technol., Haifa, Israel

Journal: J. of Guidance, Control, and Dynamics, 1994 v.17, no.5 p. 897-902

Publication Date: Sept.-Oct. 1994 Country of Publication: USA

Abstract: Two-dimensional optimal avoidance of a proportionally guided coasting missile of first-order dynamics by a constant speed aircraft is analyzed. This model allows us to investigate the "missile outrunning" and "end-game evasion" maneuvers in the same engagement. Three regions of different optimal missile avoidance strategies are identified. All strategies are based on some compromise between the principles of "missile outrunning", bleeding energy from the missile and minimizing the closing velocity and the principles of optimal "end-game evasion", a final maneuver of a critical duration perpendicular to the line of sight. A synergetic interaction between aerodynamic drag and guidance dynamics increases the sensitivity of a missile guided by proportional navigation to evasive target maneuvers. Because of the missile aerodynamic drag, the aircraft can reduce missile maneuverability by an "outrunning" maneuver so that in the "end-game" a larger miss distance can be generated. The fresh insight gained by this investigation provides important cues for both aircraft and missile designers. (14 Refs)

Title: Game optimal guidance law synthesis for short range missiles

SHINAR, J. (Technion - Israel Institute of Technology, Haifa); GUELMAN, M. (Rafael

Armament Development Authority, Haifa, Israel); GREEN, A.

J. of Guidance, Control, and Dynamics, v. 15, Jan.-Feb. 1992, p. 191-197.

Country of Origin: Israel Country of Publication: United States

Documents available from AIAA Technical Library

A horizontal pursuit-evasion game of kind in the atmosphere between a coasting pursuer with a final velocity constraint and a maneuvering evader of constant speed is considered. For this model, which is suitable to describe short range missile engagements, the adjoint equations can be integrated analytically. This allows the optimal strategies of the players on the boundary of the capture set, called the barrier, to be determined as a function of the current and final values of the state variables. The main effect of the pursuer's final velocity constraint is a substantial reduction of the capture zone. However, based on this game solution, a feedback guidance law, suitable for real-time implementation, can be synthesized and compared to other guidance laws. The results show that the corresponding capture set is much larger than the firing envelope of a similar missile guided by proportional navigation with the same final velocity constraint. (Author)

Title: Game theory for automated maneuvering during air-to-air combat

AUSTIN, FRED; CARBONE, GIRO; HINZ, HANS (Grumman Corporate Research Center, Bethpage, NY); LEWIS, MICHAEL (NASA, Ames Research Center, Moffett Field, CA); FALCO, MICHAEL

Grumman Aerospace Corp., Bethpage, NY.

(AIAA Guidance, Navigation and Control Conference, Monterey, CA, Aug. 17-19, 1987, Technical Papers. Volume 1, p. 659-669)

Journal of Guidance, Control, & Dynamics, v. 13, Nov.-Dec. 1990, p. 1143-1149.

Title: Avoidance of detection in 3-D. Pursuit-evasion differential games, III.

Authors: Zheleznov, V. S., Ivanov, M. N., Kurskii, E. A., Maslov, E. P.

Author Affiliation: Institute of Control Sciences, 117806 Moscow, Russia

Computers &Mathematics with Applications. An International Journal, 1993, 26, no. 6, 55--66. ISSN: 0898-1221 CODEN: CMAPDK

The problem of avoidance of a moving spatial zone in 3-D is considered with the assumptions: (1) the pursuer (P) and an evader (E) are moving in 3-D, (2) E is capable of instantaneous simple turns with velocity vector (v), and (3) P moves with constant (unit) velocity along a straight line. A spatial detection zone, which is the intersection of an ellipsoid and an elliptic cone, is attached to P. The evader's goal is to avoid the detection zone. The authors define a moving system XYZ, its origin is affixed to P and the X-axis is in the direction of P's velocity in R^3. The X- and Y-axes are located in the horizontal plane.

The problem is reduced to an optimal control problem with pay-off and terminal conditions.

The formulation of this problem is not a new one, as it considers the problem of detecting E by a constrained P carrying a radar system. Such problems were studied by the reviewer [in Proceedings of the 21st IEEE Conference on Decision and Control (Orlando, FL, 1982), 191--194, IEEE, New York, 1982; CCA 1983:11527] and by T. L. Vincent [in The theory and application of differential games (Coventry, 1974), 267--279, Reidel, Dordrecht, 1975; MR 58#26100].

Reviewer: El-Arabaty, Moustafa (Cairo)

Proceedings Reference: 94b#90108; 1 232 257

Pursuit-evasion differential games. III.

Edited by Y. Yavin and M. Pachter. Comput. Math. Appl. 26 (1993), no. 6.

Contributors: Yavin, Y.; Pachter, M.

Publ: Pergamon Press, Oxford,

1993, pp. v--x and 1--152. ISSN: 0898-1221 CODEN: CMAPDK

Pursuit-evasion differential games,; Special Issue: Pursuit-evasion differential games, 3 3

The twelve papers in this collection include the following: D. Ghose and U. R. Prasad, Determination of rational strategies for players in two-target games (1--11); E. A. Galperin, The cubic algorithm for global games with application to pursuit-evasion games (13--31); M.

Guelman, Control strategies in a planar pursuit evasion game with energy constraints (33--41); T. Miloh, M. Pachter and A. Segal, The effect of a finite roll rate on the miss-distance of a bank-to-turn missile (43--54); V. S. Zheleznov, M. N. Ivanov, E. A. Kurskii and E. P. Maslov, Avoidance of detection in 3-D (55--66); M. N. Ivanov and E. P. Maslov, A problem of avoidance of a rotating segment (67--75); D. A. Klementiev, The 3-D detection problem of an evader moving in a fixed plane (77--85); Y. Yavin, Applications of stochastic differential games to the suboptimal design of pulse motors (87--95); A. Garcia-Ortiz, J. Wootton, E. Y. Rodin et al., Application of semantic control to a class of pursuer-evader problems (97--124); F. Imado and T. Ishihara, Pursuit-evasion geometry analysis between two missiles and an aircraft (125--139); E. Altman [Eitan Meir Altman] and G. Koole, Stochastic scheduling games with Markov decision arrival processes (141--148).

Title: Nonlinear Regulation and Nonlinear H_∞ Control Via the State-Dependent Riccati Equation Technique.

Authors: James R. Cloutier, Cristopher N. D'Souza, and Curtis Mracek

Submitted for Publication: July 1995

Abstract: A new technique for systematically designing nonlinear regulators is introduced. The method consists of first using direct parameterization to bring the nonlinear plant to a linear structure having state-dependent coefficients (SDC). A state-dependent Riccati equation (SDRE) is then solved at each point x along the trajectory to obtain a nonlinear feedback controller of the form $u = -R^{-1}(x)g(x)^T P(x)x$, where P(x) is the solution of the SDRE. In the case of scalar x, it is shown that the SDRE approach yields the optimal solution of the nonlinear regulator. In the multivariable case, it is shown that for any SDC parameterization that is strongly controllable and strongly observable, the SDRE method produces a closed loop solution that is globally asymptotically stable provided that the state and control weighting matrices are chosen properly. It is shown that, if it exists, the parameter-dependent SDC parameterization can be computed such that the multivariable SDRE closed loop solution is optimal. Additionally, for the case of parameter variations, the robustness of the method is

characterized. A general nonlinear minimum-energy (nonlinear H_∞) problem is then posed. For this problem, the SDRE method involves the solution of two coupled SD Riccati equations at each point x along the trajectory. In the case of full state information, it is shown that the SDRE nonlinear H_∞ controller, assuming properly chosen weighting, is internally stable. Examples are provided which illustrate the effectiveness of the SDRE technique.

```
/*   missile.c      Draws the intercept surface of an aircraft and missile
 *    August 06, 1995     Mike Elgersma
 *    November 05, 1995   Include option to make 2 surface-components tangent.
 *    November 19, 1995   Put in acceleration constraints.
 *    November 19, 1995   Replaced vector iteration with faster scalar iteration.
 *    November 21, 1995   x_minus_x1_unit = [V1_unit,V1_perp]*[ce1,se1*sz1,se1*cz1]
 *  Algorithms from: Morton & Elgersma ECALM paper
 *  OpenGL code modified from: Feb. 1995 MS Journal   pp 19-40
 *-------------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>

#define pi 3.1416f/* slightly bigger than pi, so surface has a little overlap.  */
/*  Typically choose num_eta=30 and num_zeta=60 since zeta has twice the range,
    but for checking the point where two surface components meet, the surface gets
    stretched in the eta direction, near eta=0, so use more eta points.        */
#define num_eta    30 /* The number of points on  0 < eta  <   pi  on 2-sphere  */
#define num_zeta   60 /* The number of points on  0 < zeta < 2*pi  on 2-sphere  */
#define num_iter    4 /* number of iterations to find radius on warped 2-sphere */

void initialize_state(float xyz1_0[3],float xyz2_0[3],float V1_0[3],float V2_0[3],
                      float dx0[3], float V1[3],float V2[3]);
float cone2(float V1[3], float V2[3]);
void intercept2(float xyz1_0[3], float V1_0[3], float xyz2_0[3], float V2_0[3],
                float *r1_error, float T2_e_z[num_zeta+1][num_eta+1],
                float x_s[num_zeta+1][num_eta+1], float y_s[num_zeta+1][num_eta+1],
                float z_s[num_zeta+1][num_eta+1], float xyz_T2_max[3],
                float accel_1[num_zeta+1][num_eta+1],
                float accel_2[num_zeta+1][num_eta+1]);
void cross_prod(float cross[3], float vec1[3], float vec2[3]);

LONG WINAPI WndProc (HWND, UINT, WPARAM, LPARAM);
void SetDCPixelFormat (HDC);
void InitializeRC (void);
void DrawSurface(float x_s[num_zeta+1][num_eta+1], float y_s[num_zeta+1][num_eta+1],
                 float z_s[num_zeta+1][num_eta+1], float T2_e_z[num_zeta+1][num_eta+1],
                 float accel_1[num_zeta+1][num_eta+1], float accel2[num_zeta+1][num_eta+1],
                 float accel_1_limit, float accel_2_limit);
//void DrawScene (HDC hdc,  float vert_angle, float horz_angle);
void DrawScene (HDC hdc);

HPALETTE hPalette = NULL;

//const int textcolor[16] = {COLOR_WINDOWTEXT};  // used by SetSysColor to get black text
//const int   bkcolor[16] = {COLOR_WINDOW};      // used by SetSysColor to get white background
//const COLORREF blackcolor[3] = {RGB(0,0,0)};   // used by SetSysColor to get black text
//const COLORREF whitecolor[3]={RGB(255,255,255)};// used by SetSysColor to get white background

float x_s[num_zeta+1][num_eta+1]; // x coord of point on intercept surface
float y_s[num_zeta+1][num_eta+1]; // y coord of point on intercept surface
float z_s[num_zeta+1][num_eta+1]; // z coord of point on intercept surface
float T2_e_z[num_zeta+1][num_eta+1];
float accel_1[num_zeta+1][num_eta+1], accel_2[num_zeta+1][num_eta+1];// accel of veh 1 and 2
float accel_1_limit, accel_2_limit;
float xyz_T2_max[3]; /* location on surface with max intercept time */
float r1_error = 0.1f;/* max error in computed dist from aircraft to surface */
float eta2_k1;

/* The following temporary variables are used to update the state variables */
float Time = 0.0f, dT = .5f;
float center1[3], v1_0;
float center2[3], v2_0;
float cross1[3], dist_sq1, radius1, W1[3],w1, rad_vec1[3],rad_vec1_[3], cs1,ss1;
float cross2[3], dist_sq2, radius2, W2[3],w2, rad_vec2[3],rad_vec2_[3], cs2,ss2;

/* The following "state" variables get updated when time advances */
//-------------------------------------------------------------------
//
/*
```

```
//This data is planar and gives a surface that just touches the cone
float xyz1_0[3] = { 0.000f, 0.000f, 0.000f};  //  position of aircraft
//float xyz2_0[3] = {-4.603f,-8.876f, 2.954f};  //  position of missile
float   xyz2_0[3] = {-4.603f*1.4f,-8.876f*1.4f, 2.954f*1.4f};  //  position of missile(scale)
float V1_0[3] = { 1.234f,  -.593f,  .139f}; //  velocity of aircraft
float V2_0[3] = {-4.199f,  2.961f, -.761f}; // V2 >> V1   to allow intercept
float mid_point[3] = {-2.f, -4.f, 1.5f}; // center between xyz1_0 and xyz2_0 INITIAL
//
float vert_angle = -50.0f; // degrees   rotate view about vertical axis
float horz_angle = 210.0f; // degrees   rotate view about horizontal axis
// limits
float accel_1_limit = 1.4f; // 2*V1*V1/|r1-r2|
float accel_2_limit = 4.0f; // 2*V2*V2/|r1-r2|
float d_accel = .1f;      // changes in accel limits when F7 and F8 key used
*/
//
//-----------------------------------------------------------------------
// This data in the Ecalm Final Report, December 1995
float xyz1_0[3] = { 0.0f, 0.0f, 0.0f};  //  position of aircraft
float xyz2_0[3] = { 6.0f,-5.0f, 4.0f};  //  position of missile
float V1_0[3] = { -.3f, -.1f,  -.1f}; //  velocity of aircraft
float V2_0[3] = { -.2f,  .0f,  -.6f}; // V2 >> V1   to allow intercept
float mid_point[3] = {3.f, -2.5f, 2.f}; // center between xyz1_0 and xyz2_0 INITIAL
//
float vert_angle = -20.0f; // degrees   rotate view about vertical axis
float horz_angle =  20.0f; // degrees   rotate view about horizontal axis
// limits
float accel_1_limit = .1f; // 2*V1*V1/|r1-r2|
float accel_2_limit = .15f; // 2*V2*V2/|r1-r2|
float d_accel = .02f;      // changes in accel limits when F7 and F8 key used
//-----------------------------------------------------------------------
/*
// This data gies an accel2_limit region with a hole in it
float xyz1_0[3] = { 0.0f, 0.0f, 0.0f};  //  position of aircraft
float xyz2_0[3] = { 6.0f*2.0f,-5.0f*2.0f, 4.0f*2.0f};  //  position of missile
float V1_0[3] = { -.30f,   .25f,  -.20f}; //  velocity of aircraft
float V2_0[3] = { -.66f,  .51f,  -.44f}; // V2 >> V1   to allow intercept
float mid_point[3] = {3.f, -2.5f, 2.f}; // center between xyz1_0 and xyz2_0 INITIAL
//
float vert_angle = -20.0f; // degrees   rotate view about vertical axis
float horz_angle = -30.0f; // degrees   rotate view about horizontal axis
// limits
float accel_1_limit = 0.220f/2.0f; // 2*V1*V1/|r1-r2|
float accel_2_limit = 0.053f/2.0f; // 2*V2*V2/|r1-r2|
float d_accel = .002f;      // changes in accel limits when F7 and F8 key used
*/
//-----------------------------------------------------------------------
/*
// This data needs more grid points or more iterations to converge, 120x120 grid, 4 iterations
// NONplanar (almost 3 orthog vectors) and gives a surface that just touches the cone
float xyz1_0[3] = { 0.0f, 0.0f, 0.0f};  //  position of aircraft
//float xyz2_0[3] = { 0.2f,  6.0f,-4.0f};  //  position of missile
float xyz2_0[3]   = { 0.4f, 12.0f,-8.0f};  //  position of missile (x2-x1 stretches everthing)
//float V1_0[3] = { 2.294f,  .20f, .20f}; //  velocity of aircraft
float V1_0[3]   = { 2.290f,  .20f, .20f}; //  velocity of aircraft
float V2_0[3] = { .20f,  4.00f, 6.00f}; // V2 >> V1   to allow intercept
float mid_point[3] = {0.f, 3.f, -2.f};  // center between xyz1_0 and xyz2_0 INITIAL
//
float vert_angle = - 60.0f; // degrees   rotate view about vertical axis
float horz_angle =  120.0f; // degrees   rotate view about horizontal axis
// limits
float accel_1_limit = 3.0f; // 2*V1*V1/|r1-r2|
float accel_2_limit = 8.0f; // 2*V2*V2/|r1-r2|
float d_accel = .1f;      // changes in accel limits when F7 and F8 key used
*/
//-----------------------------------------------------------------------

int   big_cone = 0;  // initialize with no cone drawn (Home key turns on cone etc.)
int sign = 1; // "END" key  changes this to -1  to flip sign on keyboard entry.

float dx0[3]; //initial offset
float V1[3]; // initial velocity of aircraft
float V2[3]; // initial velocity of missile
```

```
// For changing V1 so solution components just touch.
float V2_unit[3], V2_per[3], d_r_unit[3], d_r[3], tmp[3], V1_dir[3], V1_unit[3];
float v2, los, v1_over_v2, eta2c, r2, V2_dot_r, V1_dir_norm, delta_r, cos_eta2c, sin_eta2c;

float dlos, cos_dlos, sin_dlos;  // to change los, by changing V2_0
float dr1_kiss, r1_kiss[3]; // to plot the (ellipsoid, cone) kiss point
//-------------------------------------------------------------
/*
 *  Function WinMain.
 */

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow)
{

    static char szAppName[]=
        "Intercept Surface: aircraft at 'apple stem',  black ball at max-time intercept";
    WNDCLASS wc;
    HWND hwnd;
    MSG msg;

    wc.style = CS_HREDRAW | CS_VREDRAW;   // Horizontal or Vertical redraw?
    wc.lpfnWndProc = (WNDPROC) WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH) (COLOR_WINDOW     + 1);   // overwritten by glClearColor later
    //wc.hbrBackground = (WHITE_BRUSH);   // overwritten by glClearColor later
    //wc.hbrText       = (HBRUSH) (COLOR_WINDOWTEXT + 1);     // Try to make black text  ERROR
    wc.lpszMenuName = NULL;
    wc.lpszClassName = szAppName;

    RegisterClass (&wc);

    //hwnd = CreateWindow (szAppName, szAppName,
    //                     WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
    //                     CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, //random
    //                     HWND_DESKTOP, NULL, hInstance, NULL);
    hwnd = CreateWindow (szAppName, szAppName,
                         WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
                         0,0,800,600, // upper_left_corner_xy and lower_right_corner_xy
                         HWND_DESKTOP, NULL, hInstance, NULL);

    ShowWindow (hwnd, nCmdShow);
    UpdateWindow (hwnd);

    initialize_state(xyz1_0,xyz2_0,V1_0,V2_0,  dx0,V1,V2);  // aircraft and missile

    //SetSysColors(1, bkcolor, whitecolor); //  white background
    //SetSysColors(1, textcolor, blackcolor); // black text

    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    return msg.wParam;
}

/*
 * WndProc processes messages to the main window.
 */

 LONG WINAPI WndProc (HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
 {
    static HDC hdc;
    static HGLRC hrc;
    PAINTSTRUCT ps;
    GLdouble gldAspect;
    GLsizei glnWidth, glnHeight;
    static BOOL bUp = TRUE;
```

```
static UINT nTimer;
int n;

//SetBkColor(  hdc, GetSysColor (COLOR_WINDOW) ); // see p 223
//SetTextColor(hdc, GetSysColor (COLOR_WINDOWTEXT) ); // see p 223

switch (msg) {

case WM_KEYDOWN:
    switch (wParam)
      {
      /* Rotate the view */
      case VK_LEFT:
        vert_angle -= 10.0f;
        return 0;
      case VK_RIGHT:
        vert_angle += 10.0f;
      case VK_UP:
        horz_angle -= 10.0f;
        return 0;
      case VK_DOWN:
        horz_angle += 10.0f;
        return 0;

      case VK_END:
        sign = -sign;  // Changes sign on other keyboard entries
        return 0;

      /* Change V1_ to make the two solution components just touch */
      case VK_HOME:
      if ( big_cone == 2)
        big_cone = 0;
      else
        {
        big_cone = 2;
        v2 = (float) sqrt(V2[0]*V2[0]+V2[1]*V2[1]+V2[2]*V2[2]);
        V2_dot_r = V2_0[0]*(xyz1_0[0]-xyz2_0[0]) +
                   V2_0[1]*(xyz1_0[1]-xyz2_0[1]) +
                   V2_0[2]*(xyz1_0[2]-xyz2_0[2]);
        delta_r = (float) sqrt((xyz2_0[0]-xyz1_0[0])*(xyz2_0[0]-xyz1_0[0]) +
                               (xyz2_0[1]-xyz1_0[1])*(xyz2_0[1]-xyz1_0[1]) +
                               (xyz2_0[2]-xyz1_0[2])*(xyz2_0[2]-xyz1_0[2]) );
        los = (float) acos(V2_dot_r/(v2*delta_r));

        v1_over_v2 = .724f*(1.03f - (float)sin(los/2.0f));// solution components just merge
        v1_over_v2 = .724f*( .99f - (float)sin(los/2.0f));// solution components just separate
        v1_over_v2 = .724f*(1.01f - (float)sin(los/2.0f));// solution components about kiss
        eta2c = (3.14159f - (float) asin(v1_over_v2) + los)/2.0f;  // use to draw cone

        // iterate to get better than 2% accuracy on max v1/v2 that gives separate sol comp
        v1_over_v2 = (float) ( (sin(eta2c)/eta2c) /
                 sqrt( 1 + (1/eta2c - 1/tan(eta2c))*(1/eta2c - 1/tan(eta2c)) )  );
        eta2c = (3.14159f - (float) asin(v1_over_v2) + los)/2.0f;
        v1_over_v2 = (float) ( (sin(eta2c)/eta2c) /
                 sqrt( 1 + (1/eta2c - 1/tan(eta2c))*(1/eta2c - 1/tan(eta2c)) )  );
        eta2c = (3.14159f - (float) asin(v1_over_v2) + los)/2.0f;  // use to draw cone


        d_r[0] = (xyz2_0[0]-xyz1_0[0]);
        d_r[1] = (xyz2_0[1]-xyz1_0[1]);
        d_r[2] = (xyz2_0[2]-xyz1_0[2]);
        cross_prod(tmp, d_r, V2_0);
        cross_prod(V1_dir, d_r, tmp);// V1 perp to d_r, in plane of d_r and V2_

        V1_dir_norm=(float)sqrt(V1_dir[0]*V1_dir[0]+V1_dir[1]*V1_dir[1]+V1_dir[2]*V1_dir[2]);

        V1_unit[0] = V1_dir[0]/V1_dir_norm;
        V1_unit[1] = V1_dir[1]/V1_dir_norm;
        V1_unit[2] = V1_dir[2]/V1_dir_norm;

        dr1_kiss = delta_r * (float) tan(eta2c - los);
        r1_kiss[0] = dr1_kiss*V1_unit[0];
        r1_kiss[1] = dr1_kiss*V1_unit[1];
```

```
    r1_kiss[2] = dr1_kiss*V1_unit[2];  // use to plot the (ellipsoid, cone) kiss point

    V1_0[0] = v1_over_v2*v2*V1_unit[0];
    V1_0[1] = v1_over_v2*v2*V1_unit[1];
    V1_0[2] = v1_over_v2*v2*V1_unit[2];

    Time = 0.0f;      // Restart time (then reset the initial condition)
    V1[0] = V1_0[0]; // Transfer the current state to the new initial condition
    V1[1] = V1_0[1];
    V1[2] = V1_0[2];
    V2[0] = V2_0[0];
    V2[1] = V2_0[1];
    V2[2] = V2_0[2];
    dx0[0] = xyz2_0[0] - xyz1_0[0];
    dx0[1] = xyz2_0[1] - xyz1_0[1];
    dx0[2] = xyz2_0[2] - xyz1_0[2];
    eta2_k1 = cone2(V1,V2);
    }
    return 0;

/* change the velocity of the missile */
case VK_F3:
    V1_0[0] += sign * .1f;  // increment one element of the current state
    Time = 0.0f;      // Restart time (then reset the initial condition)
    V1[0] = V1_0[0]; // Transfer the current state to the new initial condition
    V1[1] = V1_0[1];
    V1[2] = V1_0[2];
    V2[0] = V2_0[0];
    V2[1] = V2_0[1];
    V2[2] = V2_0[2];
    dx0[0] = xyz2_0[0] - xyz1_0[0];
    dx0[1] = xyz2_0[1] - xyz1_0[1];
    dx0[2] = xyz2_0[2] - xyz1_0[2];
    eta2_k1 = cone2(V1,V2);
    return 0;
case VK_F4:
    V1_0[1] += sign * .1f;
    Time = 0.0f;
    V1[0] = V1_0[0];
    V1[1] = V1_0[1];
    V1[2] = V1_0[2];
    V2[0] = V2_0[0];
    V2[1] = V2_0[1];
    V2[2] = V2_0[2];
    dx0[0] = xyz2_0[0] - xyz1_0[0];
    dx0[1] = xyz2_0[1] - xyz1_0[1];
    dx0[2] = xyz2_0[2] - xyz1_0[2];
    eta2_k1 = cone2(V1,V2);
    return 0;
case VK_F5:
    V1_0[2] += sign * .1f;
    Time = 0.0f;
    V1[0] = V1_0[0];
    V1[1] = V1_0[1];
    V1[2] = V1_0[2];
    V2[0] = V2_0[0];
    V2[1] = V2_0[1];
    V2[2] = V2_0[2];
    dx0[0] = xyz2_0[0] - xyz1_0[0];
    dx0[1] = xyz2_0[1] - xyz1_0[1];
    dx0[2] = xyz2_0[2] - xyz1_0[2];
    eta2_k1 = cone2(V1,V2);
    return 0;

case VK_F6:  // change line-of-sight angle from missile to aircraft

    // compute OLD los angle
    v2 = (float) sqrt(V2[0]*V2[0]+V2[1]*V2[1]+V2[2]*V2[2]);

    v2 = (float) sqrt(V2[0]*V2[0]+V2[1]*V2[1]+V2[2]*V2[2]);
    V2_dot_r = V2_0[0]*(xyz1_0[0]-xyz2_0[0]) +
               V2_0[1]*(xyz1_0[1]-xyz2_0[1]) +
               V2_0[2]*(xyz1_0[2]-xyz2_052]);
```

```
    delta_r = (float) sqrt((xyz2_0[0]-xyz1_0[0])*(xyz2_0[0]-xyz1_0[0]) +
                           (xyz2_0[1]-xyz1_0[1])*(xyz2_0[1]-xyz1_0[1]) +
                           (xyz2_0[2]-xyz1_0[2])*(xyz2_0[2]-xyz1_0[2]) );
    los = (float) acos(V2_dot_r/(v2*delta_r));

    // Compute unit vector perp to V2, in plane of V2 and (r2-r1)
    V2_unit[0] = V2_0[0]/v2;
    V2_unit[1] = V2_0[1]/v2;
    V2_unit[2] = V2_0[2]/v2;
    d_r_unit[0] = (xyz1_0[0]-xyz2_0[0])/delta_r;
    d_r_unit[1] = (xyz1_0[1]-xyz2_0[1])/delta_r;
    d_r_unit[2] = (xyz1_0[2]-xyz2_0[2])/delta_r;
    V2_per[0] = (d_r_unit[0] - (float)cos(los)*V2_unit[0]) / (float)sin(los);
    V2_per[1] = (d_r_unit[1] - (float)cos(los)*V2_unit[1]) / (float)sin(los);
    V2_per[2] = (d_r_unit[2] - (float)cos(los)*V2_unit[2]) / (float)sin(los);

    dlos = sign * .01f; // los  changed by the F6 key
    los += dlos;

    // Rotate V2 by the increment in the los angle
    cos_dlos = (float) cos(dlos);
    sin_dlos = (float) sin(dlos);
    V2_0[0] = V2_0[0]*cos_dlos - v2*V2_per[0]*sin_dlos;
    V2_0[1] = V2_0[1]*cos_dlos - v2*V2_per[1]*sin_dlos;
    V2_0[2] = V2_0[2]*cos_dlos - v2*V2_per[2]*sin_dlos;

    Time = 0.0f;
    V1[0] = V1_0[0];
    V1[1] = V1_0[1];
    V1[2] = V1_0[2];
    V2[0] = V2_0[0];
    V2[1] = V2_0[1];
    V2[2] = V2_0[2];
    dx0[0] = xyz2_0[0] - xyz1_0[0];
    dx0[1] = xyz2_0[1] - xyz1_0[1];
    dx0[2] = xyz2_0[2] - xyz1_0[2];
    eta2_k1 = cone2(V1,V2);
    return 0;

  case VK_F7: /* change aircraft acceleration limits */
    accel_1_limit = accel_1_limit + sign*d_accel;
    return 0;

  case VK_F8: /* change missile acceleration limits */
    accel_2_limit = accel_2_limit + sign*d_accel;
    return 0;

  /* update the state */
  case VK_F1: /* missile and aircraft each on arcs to xyz_T2_max  */
    Time += dT;
    // aircraft ------------------------------------------------------//
    v1_0 =(float) sqrt(V1_0[0]*V1_0[0]+V1_0[1]*V1_0[1]+V1_0[2]*V1_0[2]);
    cross1[0] = V1_0[1]*(xyz_T2_max[2] - xyz1_0[2]) -
                V1_0[2]*(xyz_T2_max[1] - xyz1_0[1]);
    cross1[1] = V1_0[2]*(xyz_T2_max[0] - xyz1_0[0]) -
                V1_0[0]*(xyz_T2_max[2] - xyz1_0[2]);
    cross1[2] = V1_0[0]*(xyz_T2_max[1] - xyz1_0[1]) -
                V1_0[1]*(xyz_T2_max[0] - xyz1_0[0]);
    dist_sq1 = (xyz_T2_max[0] - xyz1_0[0])*(xyz_T2_max[0] - xyz1_0[0]) +
               (xyz_T2_max[1] - xyz1_0[1])*(xyz_T2_max[1] - xyz1_0[1]) +
               (xyz_T2_max[2] - xyz1_0[2])*(xyz_T2_max[2] - xyz1_0[2]);
    radius1 = .5f*v1_0*dist_sq1/    (float)
      sqrt(cross1[0]*cross1[0]+cross1[1]*cross1[1]+cross1[2]*cross1[2]);
    W1[0] = V1_0[1]*cross1[2] - V1_0[2]*cross1[1];
    W1[1] = V1_0[2]*cross1[0] - V1_0[0]*cross1[2];
    W1[2] = V1_0[0]*cross1[1] - V1_0[1]*cross1[0];
    w1 = (float) sqrt(W1[0]*W1[0] + W1[1]*W1[1] + W1[2]*W1[2]);
    rad_vec1[0] = radius1*W1[0]/w1;
    rad_vec1[1] = radius1*W1[1]/w1;
    rad_vec1[2] = radius1*W1[2]/w1;
    rad_vec1_[0] = radius1*V1_0[0]/v1_0;
    rad_vec1_[1] = radius1*V1_0[1]/v1_0;
    rad_vec1_[2] = radius1*V1_0[2]/v1_0;6
```

```
    cs1 = (float) cos(v1_0*dT/radius1);
    ss1 = (float) sin(v1_0*dT/radius1);
    center1[0] = xyz1_0[0] -   rad_vec1[0]; // center of circle
    center1[1] = xyz1_0[1] -   rad_vec1[1];
    center1[2] = xyz1_0[2] -   rad_vec1[2];
    xyz1_0[0] = center1[0] +   rad_vec1[0]*cs1 + rad_vec1_[0]*ss1;
    xyz1_0[1] = center1[1] +   rad_vec1[1]*cs1 + rad_vec1_[1]*ss1;
    xyz1_0[2] = center1[2] +   rad_vec1[2]*cs1 + rad_vec1_[2]*ss1;
    V1_0[0] = (v1_0/radius1)*(-rad_vec1[0]*ss1 + rad_vec1_[0]*cs1);
    V1_0[1] = (v1_0/radius1)*(-rad_vec1[1]*ss1 + rad_vec1_[1]*cs1);
    V1_0[2] = (v1_0/radius1)*(-rad_vec1[2]*ss1 + rad_vec1_[2]*cs1);
    // missile ------------------------------------------------------//
    v2_0 =(float) sqrt(V2_0[0]*V2_0[0]+V2_0[1]*V2_0[1]+V2_0[2]*V2_0[2]);
    cross2[0] = V2_0[1]*(xyz_T2_max[2] - xyz2_0[2]) -
                V2_0[2]*(xyz_T2_max[1] - xyz2_0[1]);
    cross2[1] = V2_0[2]*(xyz_T2_max[0] - xyz2_0[0]) -
                V2_0[0]*(xyz_T2_max[2] - xyz2_0[2]);
    cross2[2] = V2_0[0]*(xyz_T2_max[1] - xyz2_0[1]) -
                V2_0[1]*(xyz_T2_max[0] - xyz2_0[0]);
    dist_sq2 = (xyz_T2_max[0] - xyz2_0[0])*(xyz_T2_max[0] - xyz2_0[0]) +
               (xyz_T2_max[1] - xyz2_0[1])*(xyz_T2_max[1] - xyz2_0[1]) +
               (xyz_T2_max[2] - xyz2_0[2])*(xyz_T2_max[2] - xyz2_0[2]);
    radius2 = .5f*v2_0*dist_sq2/    (float)
      sqrt(cross2[0]*cross2[0]+cross2[1]*cross2[1]+cross2[2]*cross2[2]);
    W2[0] = V2_0[1]*cross2[2] - V2_0[2]*cross2[1];
    W2[1] = V2_0[2]*cross2[0] - V2_0[0]*cross2[2];
    W2[2] = V2_0[0]*cross2[1] - V2_0[1]*cross2[0];
    w2 = (float) sqrt(W2[0]*W2[0] + W2[1]*W2[1] + W2[2]*W2[2]);
    rad_vec2[0] = radius2*W2[0]/w2;
    rad_vec2[1] = radius2*W2[1]/w2;
    rad_vec2[2] = radius2*W2[2]/w2;
    rad_vec2_[0] = radius2*V2_0[0]/v2_0;
    rad_vec2_[1] = radius2*V2_0[1]/v2_0;
    rad_vec2_[2] = radius2*V2_0[2]/v2_0;
    cs2 = (float) cos(v2_0*dT/radius2);
    ss2 = (float) sin(v2_0*dT/radius2);
    center2[0] = xyz2_0[0] -   rad_vec2[0]; // center of circle
    center2[1] = xyz2_0[1] -   rad_vec2[1];
    center2[2] = xyz2_0[2] -   rad_vec2[2];
    xyz2_0[0] = center2[0] +   rad_vec2[0]*cs2 + rad_vec2_[0]*ss2;
    xyz2_0[1] = center2[1] +   rad_vec2[1]*cs2 + rad_vec2_[1]*ss2;
    xyz2_0[2] = center2[2] +   rad_vec2[2]*cs2 + rad_vec2_[2]*ss2;
    V2_0[0] = (v2_0/radius2)*(-rad_vec2[0]*ss2 + rad_vec2_[0]*cs2);
    V2_0[1] = (v2_0/radius2)*(-rad_vec2[1]*ss2 + rad_vec2_[1]*cs2);
    V2_0[2] = (v2_0/radius2)*(-rad_vec2[2]*ss2 + rad_vec2_[2]*cs2);
    return 0;

  case VK_F2: /* missile on arc to xyz_T2_max, aircraft flies straight */
    Time += dT;
    // aircraft ------------------------------------------------------//
    xyz1_0[0] += dT*V1_0[0];
    xyz1_0[1] += dT*V1_0[1];
    xyz1_0[2] += dT*V1_0[2];
    // missile ------------------------------------------------------//
    v2_0 =(float) sqrt(V2_0[0]*V2_0[0]+V2_0[1]*V2_0[1]+V2_0[2]*V2_0[2]);
    cross2[0] = V2_0[1]*(xyz_T2_max[2] - xyz2_0[2]) -
                V2_0[2]*(xyz_T2_max[1] - xyz2_0[1]);
    cross2[1] = V2_0[2]*(xyz_T2_max[0] - xyz2_0[0]) -
                V2_0[0]*(xyz_T2_max[2] - xyz2_0[2]);
    cross2[2] = V2_0[0]*(xyz_T2_max[1] - xyz2_0[1]) -
                V2_0[1]*(xyz_T2_max[0] - xyz2_0[0]);
    dist_sq2 = (xyz_T2_max[0] - xyz2_0[0])*(xyz_T2_max[0] - xyz2_0[0]) +
               (xyz_T2_max[1] - xyz2_0[1])*(xyz_T2_max[1] - xyz2_0[1]) +
               (xyz_T2_max[2] - xyz2_0[2])*(xyz_T2_max[2] - xyz2_0[2]);
    radius2 = .5f*v2_0*dist_sq2/    (float)
      sqrt(cross2[0]*cross2[0]+cross2[1]*cross2[1]+cross2[2]*cross2[2]);
    W2[0] = V2_0[1]*cross2[2] - V2_0[2]*cross2[1];
    W2[1] = V2_0[2]*cross2[0] - V2_0[0]*cross2[2];
    W2[2] = V2_0[0]*cross2[1] - V2_0[1]*cross2[0];
    w2 = (float) sqrt(W2[0]*W2[0] + W2[1]*W2[1] + W2[2]*W2[2]);
    rad_vec2[0] = radius2*W2[0]/w2;
    rad_vec2[1] = radius2*W2[1]/w2;
    rad_vec2[2] = radius2*W2[2]/w2;
```

```
            rad_vec2_[0] = radius2*V2_0[0]/v2_0;
            rad_vec2_[1] = radius2*V2_0[1]/v2_0;
            rad_vec2_[2] = radius2*V2_0[2]/v2_0;
            cs2 = (float) cos(v2_0*dT/radius2);
            ss2 = (float) sin(v2_0*dT/radius2);
            center2[0] = xyz2_0[0] -   rad_vec2[0]; // center of circle
            center2[1] = xyz2_0[1] -   rad_vec2[1];
            center2[2] = xyz2_0[2] -   rad_vec2[2];
            xyz2_0[0] = center2[0] +   rad_vec2[0]*cs2 + rad_vec2_[0]*ss2;
            xyz2_0[1] = center2[1] +   rad_vec2[1]*cs2 + rad_vec2_[1]*ss2;
            xyz2_0[2] = center2[2] +   rad_vec2[2]*cs2 + rad_vec2_[2]*ss2;
            V2_0[0] = (v2_0/radius2)*(-rad_vec2[0]*ss2 + rad_vec2_[0]*cs2);
            V2_0[1] = (v2_0/radius2)*(-rad_vec2[1]*ss2 + rad_vec2_[1]*cs2);
            V2_0[2] = (v2_0/radius2)*(-rad_vec2[2]*ss2 + rad_vec2_[2]*cs2);
            return 0;
          }
        return 0;

    case WM_CREATE:
        //
        // Create a rendering context and set a timer.
        //
        hdc = GetDC (hwnd);
        SetDCPixelFormat (hdc);
        hrc = wglCreateContext (hdc);
        wglMakeCurrent (hdc, hrc);
        InitializeRC ();
        nTimer = SetTimer (hwnd, 1, 50, NULL);    // milliseconds
        return 0;

    case WM_SIZE:
        //
        // Redefine the viewing volume and viewport once the program
        // starts and again any time the window size changes.
        //
        glnWidth = (GLsizei) LOWORD (lParam);
        glnHeight = (GLsizei) HIWORD (lParam);
        gldAspect = (GLdouble) glnWidth / (GLdouble) glnHeight;

        glMatrixMode (GL_PROJECTION);
        glLoadIdentity ();
        gluPerspective (30.0, gldAspect, 0.48, 48.0);
        glViewport (0, 0, glnWidth, glnHeight);
        return 0;

    case WM_PAINT:
        //
        // Draw the scene.
        //
        BeginPaint (hwnd, &ps);
        //DrawScene (hdc, vert_angle, horz_angle);
        DrawScene (hdc);
        EndPaint (hwnd, &ps);
        return 0;

    case WM_TIMER:
        //
        //  force a repaint.
        //
        InvalidateRect (hwnd, NULL, FALSE);
        return 0;

    case WM_QUERYNEWPALETTE:
        //
        // If the program is using a color palette, realize the pallette
        // and update the client area when the window receives the input
        // focus.
        //
        if (hPalette != NULL) {
            if (n = RealizePalette (hdc))
                InvalidateRect (hwnd, NULL, FALSE);
            return n;
        }
```

```
        break;

    case WM_PALETTECHANGED:
        //
        // If the program is using a color palette, realize the palette
        // and update the colors in the client area when another program
        // realizes its palette.
        //
        if ((hPalette != NULL) && ((HWND) wParam != hwnd)) {
            if (RealizePalette (hdc))
                UpdateColors (hdc);
            return 0;
        }
        break;

    case WM_DESTROY:
        //
        // Clean up and terminate.
        //
        wglMakeCurrent (NULL, NULL);
        wglDeleteContext (hrc);
        ReleaseDC (hwnd, hdc);
        if (hPalette != NULL)
            DeleteObject (hPalette);
        KillTimer (hwnd, nTimer);
        PostQuitMessage (0);
        return 0;
    }
    return DefWindowProc (hwnd, msg, wParam, lParam);
}

/*
 * SetDCPixelFormat sets the pixel format for a device context in
 * preparation for creating a rendering context.
 *
 * Input parameters:
 *  hdc = Device context handle
 *
 * Returns:
 *  Nothing
 */

void SetDCPixelFormat (HDC hdc)
{
    HANDLE hHeap;
    int nColors, i;
    LPLOGPALETTE lpPalette;
    BYTE byRedMask, byGreenMask, byBlueMask;

    static PIXELFORMATDESCRIPTOR pfd = {
        sizeof (PIXELFORMATDESCRIPTOR),         // Size of this structure
        1,                                      // Version number
        PFD_DRAW_TO_WINDOW |                    // change to PFD_DRAW_TO_BITMAP (to print)
        PFD_SUPPORT_OPENGL |
        PFD_DOUBLEBUFFER,
        PFD_TYPE_RGBA,                          // RGBA pixel values
        24,                                     // 24 bit color
        0, 0, 0, 0, 0, 0,                       // Rbits Rshift, Gbits Gshift, Bbits Bshift
        0, 0,                                   // No alpha buffer
        0, 0, 0, 0, 0,                          // No accumulation buffer
        32,                                     // 32-bit depth buffer
        0,                                      // No stencil buffer
        0,                                      // No auxiliary buffer
        PFD_MAIN_PLANE,                         // Layer type
        0,                                      // Reserved (must be 0)
        0,  0, 0                                // No layer masks
    };

    int nPixelFormat;

    nPixelFormat = ChoosePixelFormat (hdc, &pfd);
    SetPixelFormat (hdc, nPixelFormat, &pfd);
```

```
    DescribePixelFormat (hdc, nPixelFormat, sizeof (PIXELFORMATDESCRIPTOR), &pfd);

    if (pfd.dwFlags & PFD_NEED_PALETTE) {
        nColors = 1 << pfd.cColorBits;
        hHeap = GetProcessHeap ();

        (LPLOGPALETTE) lpPalette = HeapAlloc (hHeap, 0,
                        sizeof (LOGPALETTE) + (nColors * sizeof (PALETTEENTRY)));

        lpPalette->palVersion = 0x300;
        lpPalette->palNumEntries = nColors;

        byRedMask = (1 << pfd.cRedBits) - 1;
        byGreenMask = (1 << pfd.cGreenBits) - 1;
        byBlueMask = (1 << pfd.cBlueBits) - 1;

        for (i=0; i<nColors; i++) {
            lpPalette->palPalEntry[i].peRed =
                (((i >> pfd.cRedShift) & byRedMask) * 255) / byRedMask;
            lpPalette->palPalEntry[i].peGreen =
                (((i >> pfd.cGreenShift) & byGreenMask) * 255) / byGreenMask;
            lpPalette->palPalEntry[i].peBlue =
                (((i >> pfd.cBlueShift) & byBlueMask) * 255) / byBlueMask;
            lpPalette->palPalEntry[i].peFlags = 0;
        }

        hPalette = CreatePalette (lpPalette);
        HeapFree (hHeap, 0, lpPalette);

        if (hPalette != NULL) {
            SelectPalette (hdc, hPalette, FALSE);
            RealizePalette (hdc);
        }
    }
} // end of SetDCPixelFormat

/*
 * InitializeRC initializes the current rendering context
 *
 * Input parameters:
 *  None
 *
 * Returns:
 *  Nothing
 */

void InitializeRC (void)
{
   GLfloat glfLightAmbient[] = { 0.1f, 0.1f, 0.1f, 1.0f };
   GLfloat glfLightDiffuse[] = { 0.7f, 0.7f, 0.7f, 1.0f };
   GLfloat glfLightSpecular[] = { 0.0f, 0.0f, 0.0f, 1.0f };

   //
   // Initialize state variables.
   //
   glFrontFace (GL_CCW);
   //glCullFace (GL_BACK); // eliminate lighting of the back side of the polygon
   //glEnable(GL_CULL_FACE);

   glDepthFunc (GL_LEQUAL);
   glEnable (GL_DEPTH_TEST);
   /* The following two lines allow missile and cone to be transparent,
      But the grphics slow down considerable, and the blendinhas a few glitches
   glEnable (GL_BLEND); // for alpha blending
   glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
   */
   glClearColor (1.0f, 1.0f, 1.0f, 0.0f); // set background color to white
   //glClearColor ( .8f,  .8f,  .8f, 0.0f); // set background color to white(grey)

   //
   // Add a light to the scene.
   //
   glLightfv (GL_LIGHT0, GL_AMBIENT, glfLightAmbient);
```

```
    glLightfv (GL_LIGHT0, GL_DIFFUSE, glfLightDiffuse);
    glLightfv (GL_LIGHT0, GL_SPECULAR, glfLightSpecular);
    glEnable (GL_LIGHTING);
    glEnable (GL_LIGHT0);
}

/*
 * DrawSurface draws a surface made of 4 cornered polygons
 *
 * Input parameters:
 * r1_e_z[num_zeta+1][num_eta+1]  an array of points defining the surface
 * r1_e_z[i-0][j-0]  =  Coordinates of first  corner of i,j polygon
 * r1_e_z[i-0][j-1]  =  Coordinates of second corner of i,j polygon
 * r1_e_z[i-1][j-1]  =  Coordinates of third  corner of i,j polygon
 * r1_e_z[i-1][j-0]  =  Coordinates of fourth corner of i,j polygon
 *
 * Returns:
 *  Nothing
 */
void DrawSurface(float x_s[num_zeta+1][num_eta+1], float y_s[num_zeta+1][num_eta+1],
                 float z_s[num_zeta+1][num_eta+1], float T2_e_z[num_zeta+1][num_eta+1],
                 float accel_1[num_zeta+1][num_eta+1], float accel2[num_zeta+1][num_eta+1],
                 float accel_1_limit, float accel_2_limit)
{
int i,j;
float x0,y0,z0, x1,y1,z1, x2,y2,z2, x3,y3,z3;
GLfloat glfColor[4];

   for (j=1; j<=num_eta; j++)
     {
     for (i=1; i<=num_zeta; i++)
       {
       x0 = x_s[i][j];
       y0 = y_s[i][j];
       z0 = z_s[i][j];

       x1 = x_s[i][j-1];
       y1 = y_s[i][j-1];
       z1 = z_s[i][j-1];

       x2 = x_s[i-1][j-1];
       y2 = y_s[i-1][j-1];
       z2 = z_s[i-1][j-1];

       x3 = x_s[i-1][j];
       y3 = y_s[i-1][j];
       z3 = z_s[i-1][j];

       //for(ic=0; ic<=3; ic++) glfColor[ic] = glfColors[1][ic];
       //glfColor[0] =     .4f*T2_e_z[i-1][j-1]/T2_e_z[num_zeta/2][num_eta/2];/* Red   */
       glfColor[0] = 1.0f;                                              /* Red   */
       glfColor[1] = 0.0f;                                              /* Green */
       glfColor[2] = 0.0f;                                              /* Blue  */
       glfColor[3] = 1.0f;
       if (accel_1[i][j]   >  accel_1_limit) glfColor[1] = 1.0f; // aircraft accel limit
       if (accel_2[i][j]   >  accel_2_limit) glfColor[2] = 1.0f; // missile  accel limit
       if (T2_e_z[i][j] == 0.0f)
         {glfColor[0] = 0.0f;
          glfColor[1] = 0.0f;
          glfColor[2] = 0.0f;
          glfColor[3] = 0.0f;// Transparent, because surface point didn't converge
         }
       glMaterialfv (GL_FRONT, GL_AMBIENT_AND_DIFFUSE, glfColor);

       glBegin (GL_POLYGON);
         if (j < num_eta)      // so 2 vectors below are both nonzero
           // normal = (x1,y1,z1)-(x0,y0,z0) cross (x3,y3,z3)-(x0,y0,z0)
           glNormal3f((y1-y0)*(z3-z0) - (z1-z0)*(y3-y0),
                      (z1-z0)*(x3-x0) - (x1-x0)*(z3-z0),
                      (x1-x0)*(y3-y0) - (y1-y0)*(x3-x0) );
         else    // (x3,y3,z3) = (x0,y0,z0)
           // normal = (x1,y1,z1)-(x0,y0,z0) cross (x2,y2,z2)-(x0,y0,z0)
           glNormal3f((y1-y0)*(z2-z0) - (z1-z0 11(y2-y0),
```

```
                    (z1-z0)*(x2-x0) - (x1-x0)*(z2-z0),
                    (x1-x0)*(y2-y0) - (y1-y0)*(x2-x0) );
          glVertex3f(x0,y0,z0);
          glVertex3f(x1,y1,z1);
          glVertex3f(x2,y2,z2);
          glVertex3f(x3,y3,z3);
        glEnd ();

      } /* end of i loop */
    }   /* end of j loop */
    glEnable(GL_NORMALIZE);  // To make the surface normals length 1
}       /* end of function */

/*
 * DrawScene uses OpenGL commands to draw the missile and intercept surface
 *
 * Input parameters:
 * hdc = Device context handle
 * vert_angle = vertical   viewing angle (up/down    arrow keys)
 * horz_angle = horizontal viewing angle (left/right arrow keys)
 *
 * Returns:
 *     Nothing
 */

 //void DrawScene (HDC hdc,  float vert_angle, float horz_angle)
 void DrawScene (HDC hdc)
 {
   int ic, buffer_length, fin_i;
   char buffer[200];  // for printing to screen
   GLUquadricObj *glquad;  // for the missile cylinder
   float v2_0; // length of V2_0[]
   float ci, si, v1, v2;
   float los, dot_prod, delta_r;
   float missile_length, cone_length;

   float text_xmin = -1.0f;  // corners of black background box to put text on
   float text_xmax =  1.0f;
   float text_ymin = -1.0f;
   float text_ymax =  -.5f;

   GLfloat glfBlue[] = {0.0f, 0.0f, 1.0f, 1.0f };
   GLfloat glfYellow[] = {1.0f, 1.0f, 0.0f, 1.0f };
   GLfloat glfColor[4];
   GLfloat glfColors[8][4]= {{1.0f, 0.0f, 0.0f, 0.5f},
                             {0.0f, 1.0f, 0.0f, 0.5f},
                             {0.0f, 0.0f, 1.0f, 0.5f},
                             {0.0f, 1.0f, 1.0f, 0.5f},
                             {1.0f, 0.0f, 1.0f, 0.5f},
                             {1.0f, 1.0f, 0.0f, 0.5f},
                             {1.0f, 1.0f, 1.0f, 0.5f},
                             {0.0f, 0.0f, 0.0f, 0.5f}};


   //
   // Clear the color and depth buffers.
   //
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  // black background ????

   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity ();
   glTranslatef (0.0f, 0.0f, -2.0f);
   //-----------------------------------------------------------------------

   //
   // Write TEXT and NUMBERS to screen
   //

   glPushMatrix();
   glPushAttrib(GL_CURRENT_BIT);// Prevent cursor from being advanced each draw

   glfColor[0] = 0.0f;glfColor[1] = 0.0f;glfColor[2] = 0.0f;glfColor[3] = 0.0f;
   glMaterialfv (GL_FRONT, GL_AMBIENT_AND_DIFFUSE, glfColor);
```

```
    glBegin (GL_POLYGON);  // black background to draw white text on
      glVertex3f(text_xmin, text_ymin, 0.0f);
      glVertex3f(text_xmin, text_ymax, 0.0f);
      glVertex3f(text_xmax, text_ymax, 0.0f);
      glVertex3f(text_xmax, text_ymin, 0.0f);
    glEnd ();

    // SetTextColor(hdc, RGB(255,0,0));  // RGB(255,255,255) = white    DOESN'T WORK
    // SelectObject(hdc, GetStockObject(BLACK_BRUSH));// try to make black text. DOESN'T WORK
    //SetSysColors(1, textcolor, blackcolor); //  black text.   PUT OUTSIDE LOOP
    //SetBkColor(  hdc, GetSysColor (COLOR_WINDOW) ); // see p 223
    //SetTextColor(hdc, GetSysColor (COLOR_WINDOWTEXT) ); // see p 223
    SelectObject(hdc, GetStockObject(SYSTEM_FONT));
    wglUseFontBitmaps(hdc, 0, 255, 1000); // Start at 0, 255 Glyphs, offset 1000
    glListBase(1000);
    v1 = (float) sqrt(V1[0]*V1[0]+V1[1]*V1[1]+V1[2]*V1[2]);
    v2 = (float) sqrt(V2[0]*V2[0]+V2[1]*V2[1]+V2[2]*V2[2]);
    dot_prod = V2_0[0]*(xyz1_0[0]-xyz2_0[0]) +
               V2_0[1]*(xyz1_0[1]-xyz2_0[1]) +
               V2_0[2]*(xyz1_0[2]-xyz2_0[2]);
    delta_r = (float) sqrt((xyz2_0[0]-xyz1_0[0])*(xyz2_0[0]-xyz1_0[0]) +
                           (xyz2_0[1]-xyz1_0[1])*(xyz2_0[1]-xyz1_0[1]) +
                           (xyz2_0[2]-xyz1_0[2])*(xyz2_0[2]-xyz1_0[2]) );
    los = (float) acos(dot_prod/(v2*delta_r));
    buffer_length = sprintf(buffer,
     "Time=%4.1f   x2(0)-x1(0)=(%4.1f %4.1f %4.1f)   V1(0)=(%4.1f %4.1f %4.1f)   V2(0)=(%4.1f %4.1f
%4.1f)  v2/v1=%4.2f >? %4.2f  error=%4.2g",
      Time,  dx0[0],dx0[1],dx0[2], V1[0],V1[1],V1[2], V2[0],V2[1],V2[2], v2/v1, 1.0f/(.724f*(1.01f-s
in(los/2.0f))), r1_error);
    //glColor3f(0.0f, 0.0f, 0.0f);   // tried to make text black
    //glTranslatef (0.0f,12.0f, 0.0f);  // locate cursor      DOESN'T WORK
    glCallLists(buffer_length, GL_UNSIGNED_BYTE, buffer);


    glPopAttrib();
    glPopMatrix();
    /*-------------------------------------------------------------------------*/
    //
    // Position the model relative to the viewpoint.
    //
    glTranslatef (0.0f, 0.0f, -35.0f);  // move the origen away from viewer, into the screen
    glRotatef (horz_angle, 1.0f, 0.0f, 0.0f);
    glRotatef (vert_angle, 0.0f, 1.0f, 0.0f);
    glTranslatef (-mid_point[0], -mid_point[1], -mid_point[2]);  // center midpoint
    //
    // Draw the missile
    //
    glPushMatrix();
    glTranslatef(xyz2_0[0], xyz2_0[1], xyz2_0[2]); // location of missile
    v1_0 = (float) sqrt(V1_0[0]*V1_0[0]+V1_0[1]*V1_0[1]+V1_0[2]*V1_0[2]);// v2/v1 printed
    v2_0 = (float) sqrt(V2_0[0]*V2_0[0]+V2_0[1]*V2_0[1]+V2_0[2]*V2_0[2]);
    glRotatef( (float) (-(180/pi)*atan2(V2_0[1],V2_0[2])), 1.0f, 0.0f, 0.0f);// -phi
    glRotatef( (float) ( (180/pi)*asin(V2_0[0]/v2_0)), 0.0f, 1.0f, 0.0f);    // -theta
    // Now missile along z axis (tail at xyz2_0)
    missile_length = 1.5f;
    cone_length    = 15.0f;
    glTranslatef(0.0f, 0.0f, -(missile_length + .6f)); // length    (so nose at xyz2_0)
    //------------------------
    // fins
    for(ic=0; ic<=3; ic++) glfColor[ic] = glfColors[1][ic];
    glMaterialfv (GL_FRONT, GL_AMBIENT_AND_DIFFUSE, glfColor);
    for (fin_i=0; fin_i<3; fin_i++)
    {
      ci = (float) cos(2*pi*fin_i/3);
      si = (float) sin(2*pi*fin_i/3);
      glBegin (GL_POLYGON);
        glNormal3f(si, ci, 0.0f);
        glVertex3f(ci*.2f, -si*.2f, .0f);
        glVertex3f(ci*.4f, -si*.4f, .0f);
        glVertex3f(ci*.4f, -si*.4f, .5f);
        glVertex3f(ci*.2f, -si*.2f, .7f);
      glEnd ();
    }   /* end of fin_i loop */
```

```
//-----------
for(ic=0; ic<=3; ic++) glfColor[ic] = glfColors[4][ic];
glMaterialfv (GL_FRONT, GL_AMBIENT_AND_DIFFUSE, glfColor);
glquad = gluNewQuadric();
//
gluCylinder(glquad, .2, .2, missile_length, 12,1);//BODY: name,rad1,rad2,length,nDiv,stacks
//
gluQuadricOrientation(glquad, GLU_INSIDE);
gluDisk(glquad, .0, .2, 12, 1);          // TAIL

glTranslatef(0.0f, 0.0f, missile_length);
gluQuadricOrientation(glquad, GLU_OUTSIDE);
gluCylinder(glquad, .2, .0, .6, 12, 1);   // NOSE
glTranslatef(0.0f, 0.0f, .6f); // origen is back at xyz2_0
//
gluDeleteQuadric(glquad);
//-----------------------------------------------------------------------------
glquad = gluNewQuadric();
if (big_cone == 1)
  { // draw cone with eta2 such that  (v1/v2) = sin(eta2)/eta2
  if (eta2_k1 > 1.5708)  // so half-cone is pointing in V2 direction
    {
    glTranslatef(0.0f, 0.0f,  -cone_length  );
    gluCylinder(glquad,cone_length*tan(-eta2_k1),0,cone_length,24,1);
    glTranslatef(0.0f, 0.0f,   cone_length  );
    }
  else
    {
    gluCylinder(glquad,0, cone_length*tan(eta2_k1), cone_length,24,1);
    }
  }

if (big_cone == 2)
  { // draw cone with eta2 such that  (v1/v2) = (sin(eta2)/eta2)*sin(eta2 - los)
  if (eta2c > 1.5708)    // so half-cone is pointing in V2 direction
    {
    glTranslatef(0.0f, 0.0f, -cone_length  );
    gluCylinder(glquad,cone_length*tan(-eta2c),0,cone_length,24,1);
    glTranslatef(0.0f, 0.0f,  cone_length );
    }
  else
    {
    gluCylinder(glquad,0,cone_length*tan(eta2c),cone_length,24,1);
    }
  }

//
gluDeleteQuadric(glquad);
glPopMatrix();

if (big_cone == 2)
  {
  // Draw a small sphere at (ellipsoid, cone) kiss point
  glPushMatrix();
  glTranslatef(xyz1_0[0], xyz1_0[1], xyz1_0[2] ); // location of vehicle 1
  glTranslatef(r1_kiss[0], r1_kiss[1], r1_kiss[2] ); // location of kiss point
  for(ic=0; ic<=3; ic++) glfColor[ic] = glfColors[4][ic];
  glMaterialfv (GL_FRONT, GL_AMBIENT_AND_DIFFUSE, glfColor);
  glquad = gluNewQuadric();
  //
  gluSphere(glquad, .2, 10, 10);  // : name, rad ,slices, stacks
  gluDeleteQuadric(glquad);
  glPopMatrix();
  }

// Draw a small black sphere at expected intercept point
glPushMatrix();
glTranslatef(xyz_T2_max[0], xyz_T2_max[1], xyz_T2_max[2]); // location of intercept
for(ic=0; ic<=3; ic++) glfColor[ic] = glfColors[7][ic];
glMaterialfv (GL_FRONT, GL_AMBIENT_AND_DIFFUSE, glfColor);
glquad = gluNewQuadric();
//
gluSphere(glquad, .2, 10, 10);  // : name, :141 ,slices, stacks
```

```
    gluDeleteQuadric(glquad);
    glPopMatrix();

    /*-------------------------------------------------------------*/
    //
    // Draw the intercept surface  (aircraft at indentation)
    //

    intercept2(xyz1_0,V1_0, xyz2_0,V2_0,
               &r1_error,T2_e_z, x_s,y_s,z_s, xyz_T2_max, accel_1, accel_2);
    DrawSurface(x_s, y_s, z_s, T2_e_z, accel_1, accel_2, accel_1_limit, accel_2_limit);
   /*-----------------------------------------------*/
    //
    // Render the scene in the pixel buffer
    //
    SwapBuffers (hdc);
}
/*-------------------------------------------------------------*/
    void initialize_state(float xyz1_0[3],float xyz2_0[3],float V1_0[3],float V2_0[3],
                          float dx0[3], float V1[3],float V2[3])
    {
    int ij; // to save initial aircraft,missile state
      for(ij=0; ij<=2; ij++)    // save initial aircraft,missile state
        {
        dx0[ij] = xyz2_0[ij] - xyz1_0[ij];
        V1[ij] = V1_0[ij];
        V2[ij] = V2_0[ij];
        }
    eta2_k1 = cone2(V1, V2);
    }
/*-------------------------------------------------------------*/
    float cone2(float V1[3], float V2[3])
    // Compute eta2 such that v1/v2 = sin(eta2)/eta2 = a + b*eta2 + c*eta2^2 + ...
    {
    float v1, v2, eta2_k1, y, w;
    float a,b,c,d,e,f,g;
    float     C,D,E,F,G;

    a = 1.0f;
    b = -a/(2.f*3.f);
    c = -b/(4.f*5.f);
    d = -c/(6.f*7.f);
    e = -d/(8.f*9.f);
    f = -e/(10.f*11.f);
    g = -f/(12.f*13*f);

    C = -c;
    D = 2*c*c-b*d;
    E = -5*c*(c*c-b*d) - b*b*e;
    F = 7*c*c*(2*c*c-3*b*d)+3*b*b*(d*d+2*c*e) - b*b*b*f;
    G = 42*c*c*c*(2*b*d-c*c)-28*b*b*c*(d*d+c*e)+7*b*b*b*(d*e+c*f)-b*b*b*b*g;

    v1 = (float) sqrt(V1[0]*V1[0]+V1[1]*V1[1]+V1[2]*V1[2]);
    v2 = (float) sqrt(V2[0]*V2[0]+V2[1]*V2[1]+V2[2]*V2[2]);
    y = v1/v2;
    w = (y-a)/(b*b);

    /* invert the series for y = sin(x)/x      */
    if (v1<v2) // so b*w > 0
      eta2_k1 = (float) sqrt(b*w*(1+C*w+D*w*w+E*w*w*w + F*w*w*w*w*w + G*w*w*w*w*w*w));
    return eta2_k1;
    }

/*****************************************************************************/

/* intercept2.c

   gcc intercept2.c -Wall -lm

% This program plots the intercept surface for a missile pursuing an aircraft.
%  Mike Elgersma   July 27, 1995        Mod for any V1 Aug 20, 1995
%-----------------------------------------------------------------------
```

```
% For vehicle_1 with (x in R^3) and initial velocity vector V1(0), the
% locus of points that can be gotten to at time T1, given any possible
% CONSTANT transverse acceleration is given by:
%
%          || x - x1(0) || ^2              ( || (x - x1(0)) x V1(0) || )
% S1:  ------------------------  * arctan(--------------------------) = T1
%      || (x - x1(0)) x V1(0) ||          (    (x - x1(0)) . V1(0)    )
%
%
% For vehicle_2 with (x in R^3) and initial velocity vector V2(0), the
% locus of points that can be gotten to at time T2, given any possible
% CONSTANT transverse acceleration is given by:
%
%           || x - x2(0) || ^2             ( || (x - x2(0)) x V2(0) || )
% S2:   ------------------------ * arctan(--------------------------) = T2
%       || (x - x2(0)) x V2(0) ||          (    (x - x2(0)) . V2(0)    )
%
%
% The intersection of surfaces S1 and S2 gives the curve on which the two
% vehicles could meet at time T1 = T2.
%
% The union of all such curves (union over all intersect times) gives a surface
% on which all intercepts must occur.
%-------------------------------------------------------------------------
% Use polar coordinates about x1(0),
%
%                             [  V1_0              ] [cos(eta1)           ]
% x - x1(0) = r1(eta1,zeta1)*[-------- , V1_0_perp]*[sin(eta1)*sin(zeta1)]
%                             [||V1_0||            ] [sin(eta1)*cos(zeta1)]
%
%
%       r1     eta1
% T1 = ----- --------
%      V1(0) sin(eta1)
%
%       r2     eta2
% T2 = ----- --------
%      V2(0) sin(eta2)
%
%  Set  r1_old = 0
%  For  eta1 = pi-eps to  0
%   For zeta1 =  -pi   to  pi
%
%    r12 = | x2(0) - x1(0) |
%    For iter = 1 to 4
%     r2_over_r12 = sqrt(r1_over_r12*r1_over_r12 - 2*cos_phi*r1_over_r12 + 1); //law of cosines
%     eta2 = acos( (cos_los+r1_over_r12*cos_gam) / r2_over_r12 );// V2_unit dot x_minus_x2_unit
%     k_of_eta = k0 * eta2 / sin(eta2);
%     r1_over_r12 = k_of_eta * r2_over_r12;
%    End iter
%
%   End // zeta1
%  End // eta1
%-------------------------------------------------------------------------


r12 = | x2(0) - x1(0) |
x2_minus_x1_unit = (x2(0) - x1(0)) / r12;
v2 = |V2(0)|
V2_unit = V2(0) / v2;
v1 = |V1(0)|
V1_unit = V1(0) / v1
V1_perp =
cos_los = - x2_minus_x1_unit dot V2_unit

For  eta1 = pi-eps to  0
  ce1 = cos(eta1); se1 = sin(eta1);
  k0 = (v1/v2) * sin(eta1)/eta1;

  For zeta1 =  -pi   to  pi

    cz1 = cos(zeta1); sz1 = sin(zeta1);
    x_minus_x1_unit = [V1_unit, V1_perp]*(ce1, se1*sz1, se1*cz1)
```

```
    cos_phi = x2_minus_x1_unit dot x_minus_x1_unit
    cos_gam =          V2_unit dot x_minus_x1_unit

    r1_over_r12 = 0;  // initialize iteration
    // Then iterate the following equations:
    For ik = 1 to 4
      r2_over_r12 = sqrt(r1_over_r12*r1_over_r12 - 2*cos_phi*r1_over_r12 + 1); //law of cosines
      eta2 = acos( (cos_los+r1_over_r12*cos_gam) / r2_over_r12 );// V2_unit dot x_minus_x2_unit
      k_of_eta = k0 * eta2 / sin(eta2);
      r1_over_r12 = k_of_eta * r2_over_r12;
    end // of ik loop

    // Exit iteration with:
    r1 = r1_over_r12 * r12;
    r2 = r2_over_r12 * r12;
    T2 = (r2/v2) * eta2/sin(eta2);
    x_surface = x1(0) + r1*(V1_unit*cos_eta1, V1_perp*[se1*sz1, se1*cz1]);

  end // zeta1
end // eta1
%-------------------------------------------------------------------------
*/
void intercept2(float xyz1_0[3], float V1_0[3], float xyz2_0[3], float V2_0[3],
                float *r1_error, float T2_e_z[num_zeta+1][num_eta+1],
                float x_s[num_zeta+1][num_eta+1], float y_s[num_zeta+1][num_eta+1],
                float z_s[num_zeta+1][num_eta+1], float xyz_T2_max[3],
                float accel_1[num_zeta+1][num_eta+1],
                float accel_2[num_zeta+1][num_eta+1])

{
int i, j, k;

float r1, r1_e_z[num_zeta+1][num_eta+1];
float V1_perp[3][2], cross[3], norm;
float eta1, zeta1, eta2;
float T1, T2, T2_max;

//-----------------------------------------
float r12, r2_over_r12, r1_over_r12, r1_over_r12_old;
float v1, V1_unit[3];
float v2, V2_unit[3];
float cos_los, cos_gam, cos_phi;
float se1,ce1, sz1,cz1;
float k0, k_of_eta;
float eta2c; // convergence test
float x_minus_x1_unit[3], x2_minus_x1_unit[3];

r12 = (float) sqrt( (xyz2_0[0]-xyz1_0[0]) * (xyz2_0[0]-xyz1_0[0]) +
                    (xyz2_0[1]-xyz1_0[1]) * (xyz2_0[1]-xyz1_0[1]) +
                    (xyz2_0[2]-xyz1_0[2]) * (xyz2_0[2]-xyz1_0[2]) );
x2_minus_x1_unit[0] = (xyz2_0[0]-xyz1_0[0]) / r12;
x2_minus_x1_unit[1] = (xyz2_0[1]-xyz1_0[1]) / r12;
x2_minus_x1_unit[2] = (xyz2_0[2]-xyz1_0[2]) / r12;

v2 = (float) sqrt(V2_0[0]*V2_0[0] + V2_0[1]*V2_0[1] + V2_0[2]*V2_0[2]);
V2_unit[0] = V2_0[0]/v2;
V2_unit[1] = V2_0[1]/v2;
V2_unit[2] = V2_0[2]/v2;

cos_los = - x2_minus_x1_unit[0]*V2_unit[0]
          - x2_minus_x1_unit[1]*V2_unit[1]
          - x2_minus_x1_unit[2]*V2_unit[2];

v1 = (float) sqrt(V1_0[0]*V1_0[0] + V1_0[1]*V1_0[1] + V1_0[2]*V1_0[2]);
V1_unit[0] = V1_0[0]/v1;
V1_unit[1] = V1_0[1]/v1;
V1_unit[2] = V1_0[2]/v1;

// form the first vector orthogonal to V1_0
if ( fabs(V1_0[0]) > .5*v1 )
  {
  V1_perp[0][0] = -V1_0[1]/ ( (float) sqrt(V1_0[1]*V1_0[1] + V1_0[0]*V1_0[0]) );
  V1_perp[1][0] =  V1_0[0]/ ( (float) sqrt(V1_0[1]*V1_0[1] + V1_0[0]*V1_0[0]) );
```

```
  V1_perp[2][0] =  0.0f;
  }
else
  {
  V1_perp[0][0] =  0.0f;
  V1_perp[1][0] =  V1_0[2]/ ( (float) sqrt(V1_0[2]*V1_0[2] + V1_0[1]*V1_0[1]) );
  V1_perp[2][0] = -V1_0[1]/ ( (float) sqrt(V1_0[2]*V1_0[2] + V1_0[1]*V1_0[1]) );
  }

// form the second vector orthogonal to V1_0
cross[0] = V1_0[1]*V1_perp[2][0] - V1_0[2]*V1_perp[1][0];
cross[1] = V1_0[2]*V1_perp[0][0] - V1_0[0]*V1_perp[2][0];
cross[2] = V1_0[0]*V1_perp[1][0] - V1_0[1]*V1_perp[0][0];
norm = (float) sqrt(cross[0]*cross[0]+cross[1]*cross[1]+cross[2]*cross[2]);
V1_perp[0][1] = cross[0]/norm;
V1_perp[1][1] = cross[1]/norm;
V1_perp[2][1] = cross[2]/norm;

T2_max = 0.0f;    // initialize
*r1_error = 0.0f; // initialize

for (j=0; j<=num_eta; j++)
  { /* start eta at pi, where r1=0 is the correct answer */
  eta1 = (pi * (num_eta-j) )/num_eta;
  se1 = (float) sin(eta1);
  ce1 = (float) cos(eta1);
  if ( fabs(eta1) < .0001) k0 =  v1/v2;        // sin(eta1)/eta1 = 1
  else                     k0 = (v1/v2) * (float) sin(eta1)/eta1;
  for (i=0; i<=num_zeta; i++)
    {
    zeta1 = (2*pi * i)/num_zeta;
    sz1 = (float) sin(zeta1);
    cz1 = (float) cos(zeta1);

    x_minus_x1_unit[0] = V1_unit[0]*ce1 + V1_perp[0][0]*se1*sz1 + V1_perp[0][1]*se1*cz1;
    x_minus_x1_unit[1] = V1_unit[1]*ce1 + V1_perp[1][0]*se1*sz1 + V1_perp[1][1]*se1*cz1;
    x_minus_x1_unit[2] = V1_unit[2]*ce1 + V1_perp[2][0]*se1*sz1 + V1_perp[2][1]*se1*cz1;

    cos_phi= x2_minus_x1_unit[0] * x_minus_x1_unit[0] +
             x2_minus_x1_unit[1] * x_minus_x1_unit[1] +
             x2_minus_x1_unit[2] * x_minus_x1_unit[2];
    cos_gam = V2_unit[0] * x_minus_x1_unit[0] +
              V2_unit[1] * x_minus_x1_unit[1] +
              V2_unit[2] * x_minus_x1_unit[2];

    if (j<2)
      {
      r1 = 0.0f;
      r1_over_r12 = r1 / r12;
      }
    else
      {
      r1 =   r1_e_z[i][j-1];       /* last value (comprimise speed vs diverge) */
      r1 = 0.0f; // prevent divergence
      r1 = 2*r1_e_z[i][j-1] - r1_e_z[i][j-2]; /* linear interp (speedier) */
      r1_over_r12 = r1 / r12;
      }

    for (k=1; k<=num_iter; k++)
      {
      r1_over_r12_old = r1_over_r12; // save old value for convergence test
      r2_over_r12=(float)sqrt(r1_over_r12*r1_over_r12-2*cos_phi*r1_over_r12+ 1);//law of cosines
      eta2=(float)acos((cos_los+r1_over_r12*cos_gam)/r2_over_r12);//V2_unit dot x_minus_x2_unit
      k_of_eta = k0 * eta2 / (float) sin(eta2);
      r1_over_r12 = k_of_eta * r2_over_r12;
      } /* end of k loop */

    T1 = (r1_over_r12_old*r12/v1) * eta1/ (float)sin(eta1);
    T2 = (r2_over_r12    *r12/v2) * eta2/ (float)sin(eta2);

    eta2c = (float) (3.14159 - asin(v1/v2) + acos(cos_los) ) / 2;
    //if ( (fabs(T1 - T2) < .1) & (eta2 < eta2c) )  // convergence test
    if ( fabs(T1 - T2) < .08 )  // convergence test
```

```
      {
      r1 = r1_over_r12 * r12;
      T2_e_z[i][j] = T2;
      }
    else
      {
      r1 = r1_e_z[i][j-1];  // no convergence, so use last value of radius
      T2_e_z[i][j] = 0.0f;  // no convergence flagged. Later, set color = transparent
      }

    r1_e_z[i][j] = r1;
    x_s[i][j] = xyz1_0[0] +
       r1*(V1_unit[0]*ce1 + V1_perp[0][0]*se1*sz1 + V1_perp[0][1]*se1*cz1);
    y_s[i][j] = xyz1_0[1] +
       r1*(V1_unit[1]*ce1 + V1_perp[1][0]*se1*sz1 + V1_perp[1][1]*se1*cz1);
    z_s[i][j] = xyz1_0[2] +
       r1*(V1_unit[2]*ce1 + V1_perp[2][0]*se1*sz1 + V1_perp[2][1]*se1*cz1);

    accel_1[i][j] = 2.f*eta1 * v1/T1; // angle*radius = v*T,    accel = v*v/radius
    accel_2[i][j] = 2.f*eta2 * v2/T2;

    if(T2_max < T2)
      {
      T2_max = T2;
      xyz_T2_max[0] = x_s[i][j];
      xyz_T2_max[1] = y_s[i][j];
      xyz_T2_max[2] = z_s[i][j];
      }
    if( *r1_error <        fabs(r1_over_r12-r1_over_r12_old)*r12)  // 0 if converged
      { *r1_error = (float)fabs(r1_over_r12-r1_over_r12_old)*r12; }
  } /* end of i loop */
} /* end of j loop */
} /* end of function intercept2 */

/*-------------------------------------------------------------------------------------*/

void cross_prod(float cross[3], float vec1[3], float vec2[3])

{

cross[0] = vec1[1]*vec2[2] - vec1[2]*vec2[1];
cross[1] = vec1[2]*vec2[0] - vec1[0]*vec2[2];
cross[2] = vec1[0]*vec2[1] -.vec1[1]*vec2[0];

}
```

```
% intercept_poly.m          Mike Elgersma, Blaise Morton          Sep 07, 1995
%                                                          updated  Sep 12, 1995
%  Put in examples where the two solution components just touch.    Nov 06, 1995
%  Balanced coeff in order8_poly before calling "roots".            Nov 07, 1995
%  Factored order8_poly when it was the square of a 4th order poly  Nov 07, 1995
%  Plotted kiss point                                               Nov 20, 1995

clear
clg
h = figure('PaperPosition',[0.5,0.5,8,10]);   % So "print" gives large figure
dataset = 13;
if (dataset==0)
  %=========================================================================
  % If [u,v,w]=[1,1,1] is a solution, then a+b=sqrt(3)=d+e+f and k*k=1
  % a = rand(1,1); b = sqrt(3)-a;                              % Cone_1 coeff
  % de = rand(2,1); def=[de;sqrt(3)-[1,1]*de]                  % Cone_2 coeff
  % k = 1;                                                     % Ellipsoid coeff
  %=========================================================================
elseif (dataset==1)
  % Get all 8 solutions REAL for the following data.
  % Only 2 solutions satisfy the HALFcone restriction to within ||err|| < .06
  % Must comment out the recomputation of eta1 and eta2 inside the ii,jj loops.
  ab0_0 = [6; 14; 0];        % Cone_1 coeff.   If a*b=0, then poly(u^2,v^2,w^2)
  def_0 = [3;  4; 7];        % Cone_2 coeff
  k0 = 1.5;                   % Ellipsoid coeff
  r1_ = [ 1;0;0];
  r2_ = [-1;0;0];
  eta1 = acos(1/norm(ab0_0));
  eta2 = acos(1/norm(def_0));
  V1_ = ab0_0; % times any scalar
  V2_ = (def_0/norm(def_0)) * (norm(V1_)*eta2*sin(eta1)) / ( k0*eta1*sin(eta2));
  %-----------------------------------
elseif (dataset==2)
  % Two eta half-cones intercept in 2 ~circles
  r1_ = [ 1;0;0];
  r2_ = [-1;0;0];
  V1_ = [-1; .1; .3 ];
  V2_ = [ 3; .2; .1 ];
  %-----------------------------------
% Note that datasets 3,4,5,6 have V1_, V2_, and r2_ - r1_  nearly orthogonal
elseif (dataset==3)
  % Solution-Sphere just smoothly merges with Solution-half-cone
  r1_ = [0;0;0];
  r2_ = [.10; 3.; -2 ];
  V1_ = [1.15; .1; .1 ];  % |V2|/|V1| = 3.113 < pi
  V2_ = [.10; 2.; 3.];
  %-----------------------------------
elseif (dataset==4)
  % Solution-Sphere just touches Solution-half-cone
  r1_ = [0;0;0];
  r2_ = [.10; 3.; -2 ];
  V1_ = [1.147; .1; .1 ];  % |V2|/|V1| = 3.121 < pi
  V2_ = [.10; 2.; 3.];
  %-----------------------------------
elseif (dataset==5)
  % Solution-Sphere just misses Solution-half-cone
  r1_ = [0;0;0];
  r2_ = [.10; 3.; -2 ];
  V1_ = [1.144; .1; .1 ];  % |V2|/|V1| = 3.129 < pi
  V2_ = [.10; 2.; 3.];
  %-----------------------------------
elseif (dataset==6)
  % Solution-Sphere just misses Solution-half-cone
  r1_ = [0;0;0];
  r2_ = [.10; 3.; -2 ];
```

```
   V1_ = [1.14; .1; .1 ];  % |V2|/|V1| = 3.140 < pi
   V2_ = [.10; 2.; 3.];
   %-----------------------------------
 elseif (dataset==7)
   % Get 4 real and 4 complex solutions.
   % The 4 real solutions satisfy the HALFcone restriction.
   % Must comment out the recomputation of eta1 and eta2 inside the ii,jj loops.
   ab0_0 = [-.6; 1.0; .0 ];   % Cone_1 coeff.  If a*b=0, then poly(u^2,v^2,w^2)
   def_0 = [-.3; 1.3; .001];  % Cone_2 coeff
   k0 = 1.02;                   % Ellipsoid coeff
   r1_ = [ 1;0;0];
   r2_ = [-1;0;0];
   eta1 = acos(1/norm(ab0_0));
   eta2 = acos(1/norm(def_0));
   V1_ = ab0_0; % times any scalar
   V2_ = (def_0/norm(def_0)) * (norm(V1_)*eta2*sin(eta1)) / ( k0*eta1*sin(eta2));
   %-----------------------------------
 elseif (dataset==8)
   % Random data in 5-dimensional set
   r1_ = [ 1;0;0];
   r2_ = [-1;0;0];
   V1_ = [rand(2,1); 0];
   V2_ = rand(3,1);
   %-----------------------------------
 elseif (dataset==9)
   % Yearly Report values used in figures 1 to 8:
   r1_ = [0;0;0];
   r2_ = [6; -5; 4];
   V1_ = [-.3; -.1; -.1];
   V2_ = [-.2;   0; -.6];
   %-----------------------------------
 elseif (dataset==10)
   % Try to find cases where "sphere" and "cone" touch
   r1_ = [ 1;0;0];
   r2_ = [-1;0;0];
    tmp =  rand(3,1);
   V1_ = [rand(2,1); 0];
   V2_ = pi*norm(V1_)*tmp/norm(tmp);  %  so v2/v1 = pi
   %-----------------------------------
 elseif (dataset==11)
   % Put slow airplane almost behind fast missile
   r1_ = [ 1;0;0]; % airplane
   r2_ = [-1;0;0]; % missile
   V1_ = [rand(2,1); 0];
   V2_ = [-3; .1; .2];
   %-----------------------------------
 elseif (dataset==12)
   % Degenerate back-substitution
   r1_ = [ 1;0;0]; % airplane
   r2_ = [-1;0;0]; % missile
   V1_ = [ 0; 1; 0];
   V2_ = [ 0; 0; 1];
   %-----------------------------------
 elseif (dataset==13)
   % Solution set components just touch when:
   % (norm(V2_)/norm(V1_)) * (sin(eta_c)/eta_c) * sin(eta_c - los) = 1
   % and eta1 = 1  at that point.
   r2_ = rand(3,1);% missile
   r1_ = rand(3,1);% airplane
   v2 = rand; junk = rand(2,1); los = 1.4
   V2_ = v2*[(r1_ - r2_)/norm(r1_ - r2_), null((r1_ - r2_)')] * ...
            [cos(los); sin(los)*junk/norm(junk)];
 %
 %  los = acos( V2_'*(r1_ - r2_) / (norm(V2_)*norm(r1_ - r2_)) )
 %
```

```
%              max     [sin(eta)                    ]
%  V1/V2 <             [-------- * sin(eta - los) ]    for separate solution
%         0<los<eta<pi [ eta                        ]                components

%             (los)        max    [sin(eta)                 ]                 los
%.724*(.99-sin(---)) <            [--------*sin(eta - los)]<.724*(1.03-sin(---))
%             ( 2 )   0<los<eta<pi[ eta                     ]                  2

   v1_over_v2=.724*(0.99-sin(los/2));%merged solution component for some unit_V1
   v1_over_v2=.724*(1.30-sin(los/2)); % separate solution components
   v1_over_v2=.724*(1.01-sin(los/2)); %
   V1_dir = skew(r2_ - r1_)*skew(r2_ - r1_)*V2_; % perp to r12 in [r12,V2] plane
   V1_dir = V1_dir + .001*rand(3,1); % planar data  (degenerate order8_poly)
   unit_V1 = V1_dir/norm(V1_dir);
  V1_ = v1_over_v2*norm(V2_)*unit_V1;
  %-----------------------------------
end

los = acos(V2_'*(r1_-r2_)/(norm(V2_)*norm(r1_-r2_)));
% sin(2*eta2c - los) = v1_over_v2   at the max
eta2c = (pi - asin(v1_over_v2) + los)/2; % angle from V2_ to touch point
ratio=(norm(V2_)/norm(V1_))*(sin(eta2c)/eta2c)*sin(eta2c-los)% > 1  separate sol

unit = (r1_ - r2_)/norm(r2_ - r1_);
Rot = [unit, null(unit')];
Rot = Rot*diag([1; 1; det(Rot)]);

stretch = 2/norm(r1_ - r2_);
orig = (r2_ + r1_)/2;
r1_s = stretch*Rot'*(r1_ - orig);
r2_s = stretch*Rot'*(r2_ - orig);
V1_s = stretch*Rot'*V1_;
V2_s = stretch*Rot'*V2_;

ca = V1_s(2)/norm(V1_s(2:3));
sa = V1_s(3)/norm(V1_s(2:3));
Rotx = [1,  0, 0;
        0, ca, sa;
        0,-sa, ca];
V1 = Rotx*V1_s; % = [xxx,yyy, 0]
V2 = Rotx*V2_s;

%-----------------------------------
eps = .0013;
ii_max = 400
jj_max = 100;
n_pts = ii_max*jj_max;
sol_set = zeros(3,8*n_pts);
max_sols = 0; % max number of real solutions that satisfy HALF-cone
num_root_mult1 = 0;
num_root_mult2 = 0;
num_root_mult4 = 0;
num_0_sols = 0;
num_1_sols = 0;
num_2_sols = 0;
num_3_sols = 0;
num_4_sols = 0;
num_5_sols = 0;
num_6_sols = 0;
num_7_sols = 0;
num_8_sols = 0;

for ii=0:ii_max-1;
ii
for jj=0:jj_max-1;
```

```
% eta2 in outer loop, since   eta2<eta2c  =>  solution component 1
eta2 = (pi)*ii/ii_max + eps; % eps avoids sin(0)/0
eta1 = (pi)*jj/jj_max + eps; %
ab0  = V1/(norm(V1)*cos(eta1)); a = ab0(1); b = ab0(2);
def  = V2/(norm(V2)*cos(eta2)); d = def(1); e = def(2); f = def(3);
k = (norm(V1)*eta2*sin(eta1))/ ...
    (norm(V2)*eta1*sin(eta2));
  m11 = d*d-1; m22 = e*e-1; m33 = f*f-1;
  m12 = d*e;   m23 = e*f;   m13 = f*d;

  Ellipsoid = diag([-1,k*k,k*k-1,k*k-1]);
%-----------------------------------------------------------------
% All polynomials are wrt to variable v.
v2 = [1, 0, 0]; v4 = [1, 0, 0, 0, 0];
v6 = [1, 0, 0, 0, 0, 0, 0]; u0v8w0 = [1, 0, 0, 0, 0, 0, 0, 0, 0];

u2 = (1/(k*k))*[(1-k*k)*b*b, (1-k*k)*2*a*b, (1-k*k)*(a*a-1) + 1];
u4 = conv(u2, u2); u6 = conv(u4, u2); u8v0w0 = conv(u4, u4);

w2 = [b*b-1, 2*a*b, a*a-1];
w4 = conv(w2, w2); w6 = conv(w4, w2); u0v0w8 = conv(w4, w4);

u6v2w0 = [u6, 0, 0];  u4v4w0 = [u4, 0, 0, 0, 0];  u2v6w0 = [u2,0,0,0,0,0,0];
u6v0w2 = conv(u6,w2); u4v2w2 = [conv(u4,w2),0,0]; u2v4w2 =[conv(u2,w2),0,0,0,0];
u0v6w2 = [w2,0,0,0,0,0,0]; u4v0w4 = conv(u4,w4);  u2v2w4 = [conv(u2,w4),0,0];
u0v4w4 = [w4,0,0,0,0]; u2v0w6 = conv(u2,w6); u0v2w6 = [w6,0,0];




order8_poly = ...
                                                    m11^4 * u8v0w0  + ...
                                                    m22^4 * u0v8w0  + ...
                                                    m33^4 * u0v0w8  + ...
                  2*(8*m12^2*(m12^2 - m11*m22) + 3*m11^2*m22^2) * u4v4w0  + ...
                  2*(8*m13^2*(m13^2 - m11*m33) + 3*m11^2*m33^2) * u4v0w4  + ...
                  2*(8*m23^2*(m23^2 - m22*m33) + 3*m22^2*m33^2) * u0v4w4  + ...
                              4*m11^2*(m11*m22 - 2*m12^2) * u6v2w0  + ...
                              4*m11^2*(m11*m33 - 2*m13^2) * u6v0w2  + ...
                              4*m22^2*(m11*m22 - 2*m12^2) * u2v6w0  + ...
                              4*m22^2*(m22*m33 - 2*m23^2) * u0v6w2  + ...
                              4*m33^2*(m22*m33 - 2*m23^2) * u0v2w6  + ...
                              4*m33^2*(m11*m33 - 2*m13^2) * u2v0w6  + ...
4*( m11^2*(3*m22*m33 - 2*m23^2) - 4*m11*(m22*m13^2 + m33*m12^2) + ...
                            8*m12*m13*(2*m11*m23 - m12*m13) ) * u4v2w2  + ...
4*( m22^2*(3*m33*m11 - 2*m13^2) - 4*m22*(m11*m23^2 + m33*m12^2) + ...
                            8*m12*m23*(2*m22*m13 - m12*m23) ) * u2v4w2  + ...
4*( m33^2*(3*m11*m22 - 2*m12^2) - 4*m33*(m11*m23^2 + m22*m13^2) + ...
                            8*m13*m23*(2*m33*m12 - m13*m23) ) * u2v2w4;

order8_poly = order8_poly / sqrt(order8_poly(1) * order8_poly(9));

% Linear poly in uv, whose coeff are poly in u2,v2,w2:
uv0_coeff = m11^2*u4 + (4*m12^2 + 2*m11*m22)*conv(u2,v2) + m22^2*v4 + ...
 (-4*m13^2+2*m11*m33)*conv(u2,w2) + (-4*m23^2+2*m22*m33)*conv(v2,w2) + m33^2*w4;
uv1_coeff = 4*(m11*m12*u2 + m12*m22*v2 - 2*m13*m23*w2 + m12*m33*w2);

v = zeros(8,1);  u = zeros(8,1);  w = zeros(8,1);
x = zeros(8,1);  y = zeros(8,1); z = zeros(8,1);
%-----------
% If the data is planar the 8th order poly factors.
% If 8th order polynomial is the square of a 4th order poly, "roots" may crash
%
```

```
[order4_poly, err_p4] = factor_poly(order8_poly);
if (err_p4 < .005)
  [order2_poly, err_p2] = factor_poly(order4_poly);
  if (err_p2 < .005)
    root_multiplicity = 4;
    v_quad = roots(order2_poly);
    for iv=1:2;
      v(4*iv-3) = v_quad(iv);
      v(4*iv-2) = v_quad(iv);
      v(4*iv-1) = v_quad(iv);
      v(4*iv-0) = v_quad(iv);
    end
  else
    root_multiplicity = 2;
    v_double = roots(order4_poly);
    for iv=1:4;
      v(2*iv-1) = v_double(iv);
      v(2*iv-0) = v_double(iv);
    end
  end
else
   root_multiplicity = 1;
   v = roots(order8_poly);
end
%-----------

if ( root_multiplicity <= 4)
  for j = 1:8
    u(j) = -polyval(uv0_coeff,v(j)) / ( v(j) * polyval(uv1_coeff,v(j)) );
  end
else
  for j = 1:4
    u(2*j-1) =  sqrt( polyval(u2,v(2*j)) );
    u(2*j  ) = -sqrt( polyval(u2,v(2*j)) );
  end
end

if ( root_multiplicity == 1)
  for j = 1:8
    % Linear poly in w, whose coeff are poly in u,v,w2
    w0_coeff = m33*polyval(w2,v(j)) + m11*u(j)^2 + 2*m12*u(j)*v(j) + m22*v(j)^2;
    w1_coeff = 2*(m13*u(j) + m23*v(j)); % w1_coeff=0 (since f=0) for planar data
    w(j) = -w0_coeff/w1_coeff;
  end
elseif ( root_multiplicity == 2)
  for j = 1:4
    w(2*j-1) =  sqrt( polyval(w2,v(2*j)) );
    w(2*j-0) = -sqrt( polyval(w2,v(2*j)) );
  end
elseif ( root_multiplicity == 4)
  for j = 1:2
    w(4*j-3) =  sqrt( polyval(w2,v(4*j)) );
    w(4*j-2) =  sqrt( polyval(w2,v(4*j)) );
    w(4*j-1) = -sqrt( polyval(w2,v(4*j)) );
    w(4*j-0) = -sqrt( polyval(w2,v(4*j)) );
  end
end

for j = 1:8
  % Bi-Rational transformation
  x(j) = (u(j)+1)/(u(j)-1);
  y(j) =   2*v(j)/(u(j)-1);
  z(j) =   2*w(j)/(u(j)-1);
end
uvw = [u,v,w];
```

```
xyz = [x,y,z];
n_sols = 0; % count how many of the 8 roots are real and satisfy HALF-cone
eps_err = .05;
for j=1:8
  vec = [x(j)-1; x(j)+1; y(j); z(j)];
  vec_1 = [x(j)-1; y(j); z(j)];
  vec_2 = [x(j)+1; y(j); z(j)];
  err4 = norm(vec_1) - k*norm(vec_2); % Ellipsoid is square of this.
  err5 = V1'*vec_1 - cos(eta1)*norm(V1)*norm(vec_1); % HALF cone_1
  err6 = V2'*vec_2 - cos(eta2)*norm(V2)*norm(vec_2); % HALF cone_2
  %disp([ sprintf('err4, err5, err6 = %g %g %g', err4, err5, err6) ])
  if( ( abs(imag(xyz(j,1))) < .02 + .02*abs(real(xyz(j,1))) ) & ...
      ( abs(imag(xyz(j,2))) < .02 + .02*abs(real(xyz(j,2))) ) & ...
      ( abs(imag(xyz(j,3))) < .02 + .02*abs(real(xyz(j,3))) ) & ...
      (norm([err4,err5,err6])<eps_err) )
    sol_set(:, j + 8*(ii*jj_max + jj) ) = xyz(j,:)';
    n_sols = n_sols + 1;
    if(root_multiplicity ==1) num_root_mult1 = num_root_mult1 + 1; end;
    if(root_multiplicity ==2) num_root_mult2 = num_root_mult2 + 1; end;
    if(root_multiplicity ==4) num_root_mult4 = num_root_mult4 + 1; end;
  else
    sol_set(:, j + 8*(ii*jj_max + jj) ) = [0;0;0];
  end
end

if(n_sols == 0) num_0_sols = num_0_sols + 1; end;
if(n_sols == 1) num_1_sols = num_1_sols + 1; end;
if(n_sols == 2) num_2_sols = num_2_sols + 1; end;
if(n_sols == 3) num_3_sols = num_3_sols + 1; end;
if(n_sols == 4) num_4_sols = num_4_sols + 1; end;
if(n_sols == 5) num_5_sols = num_5_sols + 1; end;
if(n_sols == 6) num_6_sols = num_6_sols + 1; end;
if(n_sols == 7) num_7_sols = num_7_sols + 1; end;
if(n_sols == 8) num_8_sols = num_8_sols + 1; end;

if (n_sols > max_sols)
  max_sols = n_sols
  eta1_max_sols = eta1
  eta2_max_sols = eta2
  multiplicity_sols = root_multiplicity
end

end % end of jj loop
end % end of ii loop

max_sols

  % rotate and project
  aa = -.37; bb= .34; cc =  .35;  % three small angles (radians)
  aa =  .47; bb=-.44; cc = -.45;  % three small angles (radians)
  S = [  0  cc -bb;
       -cc   0  aa;
        bb -aa   0]; % skew symmetric
  C = (eye(3)-S)/(eye(3)+S);  % Cayley Transform from R^3 to SO(3)
  sol_set_rot = C*sol_set;
  sol1 = 8*jj_max*ii_max*eta2c/pi; % solution component 1
  sol2 = 8*jj_max*ii_max;          % solution component 1
  plot(sol_set_rot(1,1:sol1),sol_set_rot(2,1:sol1), '.');
  hold on
  plot(sol_set_rot(1,1+sol1:sol2),sol_set_rot(2,1+sol1:sol2), '+');
  dr = r2_ - r1_;
  title_txt1=sprintf('x2-x1=(%5.3f %5.3f %5.3f ),  ', dr(1),dr(2),dr(3));
  title_txt2=sprintf('V1=(%5.3f %5.3f %5.3f ),  ',  V1_(1), V1_(2), V1_(3));
  title_txt3=sprintf('V2=(%5.3f %5.3f %5.3f )',  V2_(1), V2_(2), V2_(3));
  title([title_txt1, title_txt2, title_txt3])
```

```
  delta_r = norm(r1_s - r2_s) * tan ( eta2c - los);
  kiss_point = C* ( r1_s + delta_r*V1/norm(V1) ); % in screen coordinates
  plot( kiss_point(1), kiss_point(2), 'o')

  airplane_tail = C*(r1_s - V1/.2);
  airplane_nose = C*r1_s;
  plot( airplane_nose(1), airplane_nose(2), 'o');
  plot([airplane_nose(1),airplane_tail(1)],[airplane_nose(2),airplane_tail(2)]);

  missile_tail = C*(r2_s - V2/.2);
  missile_nose = C*r2_s;
  plot(  missile_nose(1),  missile_nose(2), '*');
  plot([missile_nose(1),missile_tail(1)],[missile_nose(2),missile_tail(2)]);

  % The origin is midway between the airplane, r1_s=(1,0,0) and missile(-1,0,0)
  axis([-1,2,-2,1]);
  %axis([-3,3,-3,3]);
  %axis([-5,5,-5,5]);
  %axis([-10,10,-10,10]);
  hold off

xlabel('halfcone(eta1) halfcone(eta2) ellipsoid(eta1,eta2) 0<eta1<pi 0<eta2<pi')
v2_over_v1 = norm(V2_)/norm(V1_);
v2_over_v1_touch = 1/ ( .724*(1.01-sin(los/2)) );
yt=sprintf('v2/v1 = %3.2f,    v2/v1_touch = %3.2f',v2_over_v1,v2_over_v1_touch);
ytext2 = sprintf('    los = %1.2f,   max_sols = %1.0f', los, max_sols);
ytext3 = sprintf('   eta2c = %1.3f    ', eta2c);
ytext4=sprintf('kiss_pt = x1 + %5.3f * V1_unit',delta_r/stretch);
ylabel([yt, ytext2, ytext3, ytext4])

% print

num_124_root_mult = [num_root_mult1 num_root_mult2 num_root_mult4]

real_012345678_sols = [num_0_sols num_1_sols num_2_sols num_3_sols ...
                num_4_sols num_5_sols num_6_sols num_7_sols num_8_sols]
disp('num_124_root_mult * [1;1;1] = real_012345678_sols * [0;1;2;3;4;5;6;7;8]')

save intercept_pol_dat
```

```
% v1v2_vs_los.m                                  Mike Elgersma  December 22, 1995
%---------------------------
% See "Components of the Intercept Surface" in ecalm/nonlin/yearly_report.Oct95
% See "Components of the Intercept Surface" in ecalm/nonlin/yearly_report.Dec95
%
% Cone and ellipsoid just touch (double root) when   v1/v2 = f(eta, los)
%
% For each value of los,
%   1) compute the max, wrt eta, of f(eta,los)
%   2) set max v1/v2 = max f
% Plot max v1/v2  vs each such los.
%---------------------------
% Find smallest v1/v2 that causes accel_2 to hit some constraint:
%
%           2*v2*v2*sin(eta2)
% accel_2 = -----------------      solve for  r2  and plug into next equation:
%                 r2
%
% Intercept surface just touches constant accel_2 surface at:
%
% sin(eta1)  V1          { r1 sin(eta2)}       { ||(r_1-r_2)-r2*1_r2|| sin(eta2)}
% --------- ---  =  min { -- ---------} = min { ---------------------- ---------}
%   eta1     V2    eta2 { r2   eta2   }       {              r2          eta2    }
%
%       { ||     (cos(los))      (cos(eta2)) ||            }
%       { || r12*(sin(los)) - r2*(sin(eta2)) ||  sin(eta2) }
% = min { ------------------------------------ --------- }
%       {                    r2                   eta2    }
%
% and r12/r2 = accel_2 / accel_12*sin(eta2)    where  accel_12 = 2*v2*v2/r12
%
% sin(eta1)  V1          {  ||acel_2   (1)             (cos(eta2-los)  ||
% --------   --  =  min  {  ||-------- (0) - sin(eta2)*(sin(eta2-los)) || / eta2
%   eta1     V2    eta2  {  ||accel_12                                  ||
%
%               < ( accel_2/accel_12 - sin(los) ) / los       (set eta2 = los)
%
%---------------------------
h= figure('PaperPosition',[0.5,0.5,8,10]);.  % So "print" gives large figure
epss = 1e-9;
num_los = 100;
los_min = epss; los_max = pi-epss; % 0 < los  < pi

los            = zeros(1,num_los+1);
max_v1_over_v2 = zeros(1,num_los+1);
lower_bound    = zeros(1,num_los+1);
middle         = zeros(1,num_los+1);
difference     = zeros(1,num_los+1);
upper_bound    = zeros(1,num_los+1);
eta_at_max     = zeros(1,num_los+1);
eta_at_max_    = zeros(1,num_los+1);
v1_over_v2_iter_con = zeros(1,num_los+1);
d_los = (los_max - los_min)/num_los;

accel = [.79, 1.0, 1.385];
min_v1_over_v2_accel = 1e9*ones(max(size(accel)),num_los+1);
eta_at_min_accel=zeros(max(size(accel)),num_los+1);

for i = 1:num_los+1;
  i
  los(i) = los_min + (i-1)*d_los;

  % Find max v1/v2 for separate solution components. No acceleration limit.
  for eta = los(i):.01:pi-epss/2;
    %-------------------------------------------------------------
```

```
    % max v1/v2 for separate solution components. No acceleration limit.
    v1_over_v2 = (sin(eta)/eta) * abs(sin (eta - los(i)));
    if ( v1_over_v2 > max_v1_over_v2(i) )
      max_v1_over_v2(i) = v1_over_v2;
      eta_at_max(i) = eta;
    end
    %-----------------------------------------------------------------
  end

  % Find min v1/v2 that still hits acceleration limit on the pursuer.
  % The min switches branches (of intercept surface) when v1/v2 = sin(los/2)
  % avoid the 2nd branch (with eta2=pi) by running eta2 from 0 to (los+pi)/2
  for ia = 1:max(size(accel));
  a = accel(ia);  %  accel_2/(2*V2*V2/r12);
  for eta = epss:.01:(3*los(i) + pi)/4
    %-----------------------------------------------------------------
    v1_over_v2_accel=norm([a;0]-sin(eta)*[cos(eta-los(i));sin(eta-los(i))])/eta;
    if ( v1_over_v2_accel < min_v1_over_v2_accel(ia,i) )
      min_v1_over_v2_accel(ia,i) = v1_over_v2_accel;
      eta_at_min_accel(ia,i) = eta;
    end
     %
    %-----------------------------------------------------------------
  end % end of eta loop
  if ( abs(sin(eta_at_min_accel(ia,i)))  > a ) %      accel_2 > a2  when v1=0
    min_v1_over_v2_accel(ia,i) = 0;
  end
  %
  end   % end of ia loop

  %max v1/v2  ~=  .724*( 1 - sin(los(i)/2) );
  %
  lower_bound(i) = .724*( 0.99 - sin(los(i)/2) );
  middle(i)      = .724*( 1.00 - sin(los(i)/2) );
  upper_bound(i) = .724*( 1.03 - sin(los(i)/2) );
  %
  difference(i)  = .033 * exp(-.8*los(i)) * sin(1.6*los(i));
  %difference(i)  = .015 * sin(2*los(i)) / (1+(los(i)/1.5)^6;
  eta_at_max_(i) = ( pi - asin(max_v1_over_v2(i)) + los(i) )/2;
  v1_over_v2_iter_con(i) = (sin(eta_at_max_(i))/eta_at_max_(i) ) / ...
    sqrt(1 + (1/eta_at_max_(i) - 1/tan(eta_at_max_(i)))^2 );
end

%plot(los,max_v1_over_v2, los,v1_over_v2_iter_con) % need max first
plot(los,max_v1_over_v2, los,upper_bound, los,lower_bound)
title('max v1/v2 that satisfies separate solution component bound')
ylabel('.724 ( .99 - sin(los/2) ) < v1/v2 < .724 ( 1.03 - sin(los/2) )')
xlabel('los')
axis([0,pi,0,1])
pause

% plot(los,max_v1_over_v2 - middle, los,difference)
% plot(los, max_v1_over_v2-lower_bound)
% plot(los, max_v1_over_v2-upper_bound)

% plot(los(1:num_los),eta_at_max(1:num_los)-eta_at_max_(1:num_los))

plot(eta_at_max_,max_v1_over_v2, eta_at_max_, v1_over_v2_iter_con)
ylabel('max v1/v2 separate solution.  v1/v2 iteration converge')
xlabel('eta at max')
pause

junk = zeros(max(size(accel)),num_los+1);
for ii = 1:max(size(accel))
for ij = 1:num_los+1
```

```
 %junk(ii,ij) = min(accel(ii)/pi, (accel(ii) - sin(los(ij)))/los(ij) );
  junk(ii,ij) = (accel(ii) - sin(los(ij)))/los(ij) ;
end
end

%plot(los,sin(.5*los),  los,min_v1_over_v2_accel,  los,junk, los,max_v1_over_v2)
plot(los,max_v1_over_v2, los,min_v1_over_v2_accel )
titletext1 = 'max v1/v2 to guarantee:   (Solid) Separate sol components.   ';
titletext2 = '(Dotted) accel2 < ';
titletext3 = sprintf('[%4.2f, %4.2f, %4.2f]', accel(1),accel(2),accel(3));
title([titletext1, titletext2, titletext3])
ylabel('v1/v2 = min_over_e (|[accel2;0] - sin(e)*[cos(e-los);sin(e-los)]| / e)')
xlabel('los')
axis([0,pi,0,1])

save v1v2_vs_los
```

Assessing The Impact of Estimation Errors on Guidance Algorithm Performance

Thomas L. Ting

Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN 55418

ABSTRACT

In this paper we focus on the interrelationship between state estimation filtering errors and their ultimate impact upon guidance algorithm performance. We compare the performance of proportional navigation guidance and robust guidance versus fuel optimal guidance in a simple two-dimensional missile/target engagement. We find that with perfect state estimation both proportional navigation and robust guidance achieve small miss distances and come very close to matching the ideal fuel optimum response. However, in the presence of realistic state estimation errors their performance degrades sharply. We explore the magnitude of this performance degradation and its impact on overall missile system performance.

INTRODUCTION

In a typical missle system design active or passive seeker information is input into a filter to obtain state estimates of the position, velocity and acceleration of the target. This information is then input into a guidance algorithm which uses it and a knowledge of intercept dynamics to ultimately produce a missile acceleration command profile.

From our past experience with missile system design we have noticed a clear need for understanding the impact of state estimation errors on guidance system performance. These two subsystems are typically designed independently by different engineers. The filter designer attempts to obtain the best possible state estimates, optimizing a complex tradeoff between state estimation errors, complexity of filter designs, input noise characteristics and computing power requirements. Simultaneously, the guidance law designer conducts his design assuming that he will receive near perfect state information and assuming that his algorithm is implementable by existing missile system hardware with little or no modification. His main concern is to tradeoff fuel consumption versus miss distance to achieve a satisfactory guidance law.

The main drawback of the above scenario is that although the filtering and guidance subsystems are each designed independently, the accuracy of the state estimates has a large impact upon guidance law performance.

To illustrate this impact we will compare the performance of three different guidance algorithms: fuel optimal guidance, proportional navigation guidance and robust guidance, using the simple two-dimensional guidance problem shown in Figure 1. Here, the target and the missile are represented by point mass models where the target is denoted by the letter T and the missile is denoted by the letter M. The relative positions of the missile and target are represented with respect to a xy-coordinate frame whose origin is always coincident with the location of the target. In this coordinate frame the target has zero velocity and acceleration and is always located at the origin. Conversely, the missile always begins a relative distance from the target (origin) and is also moving relative to the target (origin) with some velocity and acceleration. To further simplify our example, we assumed no gravity effects (i.e. an exoatmospheric engagement).

GUIDANCE ALGORITHMS

In this section we will briefly discuss details of the three different guidance algorithms. The first algorithm, fuel optimal guidance, is quite similar to bang-bang control and produces a theoretically achievable optimum level of performance. The second algorithm, proportion navigation guidance, is the most commonly used guidance algorithm in practice today due to its combination of good performance and easy implementation. The third algorithm, robust guidance, is based on some recently developed ideas [Ting (1992)] designed to enhance the robustness of the guidance algorithm to state estimation errors. It requires more computation than proportional navigation guidance and has not been implemented in practice. However, it has shown some good properties in some preliminary simulation studies.

Fuel Optimal Guidance

To assess the performance impact of estimation errors and different guidance algorithms on overall system performance, it is important to quantify the optimum achievable performance as a benchmark. In a realistic missile system this requires hitting the target with a minimum of fuel expenditure, while simultaneously adhereing to hardware imposed acceleration limits. To this end, consider the linear system

$$\dot{y} = v_y \tag{1}$$

$$\dot{v_y} = a \tag{2}$$

where the maximum achievable acceleration is constrained by a magnitude limit a_{max}. The fuel optimal guidance problem can now be formulated as follows: Given any initial condition $y(0) = y_o$ and $v_y(0) = v_{y_o}$ and a fixed final time t_F find the acceleration profile $a(t)$

which drives the system to the final condition $y(t_F) = 0$ and $v_y(t_F) = free$ while minimizing the cost functional

$$J = \int_0^{t_F} \| a(t) \| \, dt. \tag{2}$$

This is a standard optimal control problem which can be solved via a straightforward application of the maximum principle [Athans and Falb (1971)]. The resulting optimal control law is an 'accelerate and coast' single switch sequence of accelerations of the form $(-a_{max}, 0)$ or $(+a_{max}, 0)$ depending on where the initial condition (y_o, v_{y_o}) lies in the (y, v_y)-plane. The detailed optimal control law is given by:

Case 1: If $v_{y_o} \geq -\frac{y_o}{t_F}$ then

$$a(t) = \left\{ \begin{array}{cc} -1 & for \quad 0 \leq t \leq t_s \\ -0 & for \quad t_s \leq t \leq t_F \end{array} \right\}. \tag{3}$$

where

$$z = (v_{y_o} - a_{max} t_F) + \sqrt{(a_{max} t_F - v_{y_o}^2)^2 - (2 a_{max} y_o + v_{y_o}^2)} \tag{4}$$

and

$$t_s = \frac{v_{y_o} - z}{a_{max}} \tag{5}$$

Case 2: If $v_{y_o} = -\frac{y_o}{t_F}$ then

$$a(t) = \left\{ \begin{array}{cc} 0 & for \quad 0 \leq t \leq t_F \end{array} \right\}. \tag{6}$$

Case 3: If $v_{y_o} \leq -\frac{y_o}{t_F}$ then

$$a(t) = \left\{ \begin{array}{cc} +1 & for \quad 0 \leq t \leq t_s \\ +0 & for \quad t_s \leq t \leq t_F \end{array} \right\}. \tag{7}$$

where

$$z = (v_{y_o} + a_{max} t_F) - \sqrt{(a_{max} t_F + v_{y_o}^2)^2 - (v_{y_o}^2 - 2 a_{max} y_o)} \tag{8}$$

and

$$t_s = \frac{v_{y_o} - z}{a_{max}}. \tag{9}$$

It is clear that the fuel optimal guidance law is governed by the location of the initial condition (y_o, v_{y_o}) relative to the line

$$v_{y_o} = -\frac{y_o}{t_F} \tag{10}$$

in the (y, v_y)-plane. If the initial condition lies on this line then the optimal guidance law is zero acceleration for all time because the vehicle is already on a homing trajectory with the target. If the initial condition lies above this line then the optimal guidance law is to apply maximum negative acceleration from time t=0 until the switching time t=t_s and then to apply zero acceleration for the duration of the flight. Finally, if the initial condition lies below this line then the optimal guidance law is to apply maximum positive acceleration from time t=0 until the switching time t=t_s and then to apply zero acceleration for the duration of the flight. Essentially, the concept of this guidance law can be summarized as follows: Accelerate fully in either the positive or negative direction (whichever is appropriate) until the zero effort miss distance is zero and then coast in to hit the target.

Proportional Navigation Guidance

Proportional navigation is the most commonly used navigation algorithm in practice today. In this algorithm the missile normal acceleration command is given by

$$a_N = K \dot{\lambda} V_c \tag{11}$$

where K is a constant (typically between 2 and 5), $\dot{\lambda}$ is the rate of change of the line-of-sight angle between the missile and target and V_c is the relative closing velocity between the missile and target. Mathematically, this algorithm can be obtained by solving the following linear quadratic optimization problem [Bryson and Ho (1973)]. Consider the linear system given in Equation(1) with initial conditions $y(t_o) = y_o$ and $v_y(t_o) = v_{y_o}$. Suppose t_F is fixed and we want to find the acceleration profile $a(t)$ which minimizes the cost functional

$$J = .5(c_1 y(t_F)^2 + c_2 v_y(t_F)^2) + .5 \int_{t_o}^{t_F} a(t)^2 dt. \tag{12}$$

Given any set of constants c_1 and c_2 we can easily solve for the optimal acceleration $a^*(t)$ as a function of c_1, c_2, $y(t)$ and $v_y(t)$. If we modify the cost function J to emphasize only terminal position, i.e. we let c_1 approach infinity and let c_2 approach zero, then $a^*(t)$ reduces to the proportional navigation control law.

Proportional navigation exhibits excellent performance under ideal conditions. However, it does not address some of the implementation issues which may arise in a real life missile system application. For instance, in pro-nav guidance the desired set of normal acceleration commands is allowed to take on an unbounded, continuous set of values. Clearly, no hardware implementation can achieve infinite acceleration, so an excessively large acceleration commands can produce temporary saturations. Also, continuouly valued acceleration commands are incompatible with the increasingly common ON/OFF or discretely throttleable thrusters. This complication is often treated by implementing corrective schemes such as pulse-width modulation (PWM), to approximate a continuous signal by a sequence of discrete signals. The consequences of such saturations and approximations will of course be reflected in degraded performance.

Pro-nav guidance also does not address the issue of minimizing fuel usage under imperfect conditions. In our past experience, pro-nav guidance can be extremely sensitive to estimation errors which may result in an excessive amount of thruster activity. In simulation runs this is evident by near simultaneous firing of a thruster in opposite directions, thus nearly nullifying the effect of both firings. This phenomenon can be alleviated somewhat by passing

the pro-nav guidance command output through a deadzone filter. The size of the 'optimal' deadzone is typically determined experimentally and is dependent upon other characteristics of the overall system.

A Robust Guidance Algorithm

An approach for developing a robust guidance algorithm was introduced in [Ting (1992)]. Despite its skeleton development, this algorithm is presented because it demonstrates some of the potential performance improvements available by incorporating system integration considerations into subsystem design. The novel aspect of this algorithm is that its acceleration command output is directly affected by the magnitude of the state estimation errors.

The robust guidance approach consists of three main steps: 1) A Zero Effort Miss distance calculation, 2) A computed miss distance error radius R to account for filter estimation errors and 3) A miss distance deadzone concept. The Zero Effort Miss distance (ZEM) is computable at any point in a missile-target engagement. It represents the distance by which the missile will miss the target assuming that the missile acceleration is zero for the duration of its flight. For the robust guidance algorithm concept, it is necessary to compute not only the ZEM but also the sensitivity of the ZEM with respect to the target state estimates of position, velocity and acceleration.

The main idea behind the robust guidance approach is embodied in the miss distance deadzone concept. This concept can loosely be stated as follows. Given the computed ZEM and the error radius R, if the actual ZEM could in fact be zero then do not change the course of the missile. However, if the actual ZEM cannot be zero, then output commands which place the missile on an intercept trajectory with its target. In contrast to most guidance algorithms which output acceleration commands, the robust guidance algorithm outputs commands in the form of a velocity correction. The miss distance deadzone concept can now be restated as follows. If the magnitude of the computed ZEM is small then set the velocity correction to zero. If the magnitude of the computed ZEM is large then set the velocity correction equal to a value which should correct the ZEM to zero. In this algorithm the boundary between large and small is governed by the computed miss distance error radius R.

In this paper our robust guidance algorithm will only be required to output y-component velocity corrections. In this context, the miss distance deadzone concept can be interpreted as shown in Figure 2. Here, the values along the horizontal zxis represent various values of ZEM. For instance, suppose the computed value of ZEM $= ZEM_o$. If the computed miss distance error radius R $= R_1$ then the y-velocity component correction is set to zero. However, if the miss distance radius R $= R_2$ then the y-velocity component correction is set to $\frac{ZEM}{\frac{\Delta ZEM}{\Delta v_y}}$.

THE IMPACT OF ESTIMATION ERRORS

To assess the relative performance of various guidance algorithms we used a two-dimensional simulated engagement between a missile and a target as described earlier and as shown in Figure 1. The initial conditions of this simulation were as follows. The missile and target began 100,000 units apart in the x-direction with a relative missile/target velocity of 10,000 units/sec in the x-direction. We assumed that neither the missile nor the target were accelerating in the x-direction and thus each simulation run had a fixed duration of 10 seconds. The relative y-displacement and velocity of the missile and target were specified at the beginning of each simulation run. It was assumed that the target had zero y-acceleration and the missile y-acceleration was provided by the output of the guidance algorithm.

We conducted two separate types of analyses. The first type was called single run analysis and involved analyzing the performance of each of the guidance algorithms based on miss distance and fuel consumption for the same engagement scenario. The second type was called initial condition system analysis and involved analyzing each of the guidance algorithms based on the set of initial engagement scenarios which were hittable given a fixed set of system characteristics.

Single Run Analysis

We ran a variety of simulations of the engagement described above using the fuel optimal, proportional navigation and robust guidance algorithms described earlier. We shall describe our analysis for an engagement where the initial relative y-displacement was -3000 units and the initial relative y-velocity was 0 units/sec.

We first considered the case where the estimation filter produced perfect state estimates. This gave us a chance to assess how close the pro-nav guidance and robust guidance algorithms came to achieving the true 'optimal' solution. The results of these simulation runs are summarized in Figures 3-4. Figure 3 shows the miss distance performance of all three algorithms. Clearly, they all demonstrated acceptable performance in hitting the target. Figure 4 shows the total amount of thruster firing time for the duration of the simulation. As expected, the fuel optimal guidance produced the minimum amount of thruster firings. The robust guidance required roughly 6 percent more thruster firings while pro-nav guidance required roughly 11 percent more thruster firings.

The rationale behind the variations in fuel usage is evident in Figures 5-7 which detail each of the three guidance algorithms output acceleration commands. The optimal fuel guidance output acceleration command is shown in Figure 5. From this figure we can see that the fuel

optimal strategy is to turn on the thrusters initially for as long as necessary to put the missile on an intercept trajectory, i.e. drive ZEM to zero, and then shut them off for the remainder of the run. Heuristically, this is clearly the fuel optimal solution because a fixed amount of acceleration change at the beginning of the run will result in a much larger total position change over the course of the run than the same amount of acceleration change applied at a later time. From Figures 6 and 7 we can see that both the robust and pro-nav guidance algorithms produce nonzero acceleration commands throughout the latter stages of the run. It is interesting to note that in this case the robust guidance acceleration profile more closely matches the theoretical optimum than does the pro-nav guidance acceleration profile. This is especially true at the beginning of the run when the robust guidance law maintains full acceleration for a longer time than the pro-nav guidance law. Consequently, the ZEM associated with the robust guidance law is initially reduced faster than the ZEM associated with the pro-nav guidance law. This can be seen in Figure 4.

We next considered the case with imperfect state estimates. This was accomplished by corrupting each of the state values with random noise with zero mean and standard deviation equal to 5 percent of the nominal state value. The results of these simulation runs are summarized in Figures 8-11. As seen in Figure 8, all three guidance algorithms were still able to achieve an acceptably small miss distance. However, from Figure 9 we can see that the associated effort, i.e. thruster firings, increased sharply over the perfect state estimation case. The estimation error caused the pro-nav guidance algorithm to increase thruster firings by 40 percent while the robust guidance algorithm increased thruster firings by 23 percent. These increased firings are clearly evident in Figures 10 and 11 where we see persistent, near coincident, positive and negative thruster firings throughout the latter stages of the run. Such firings nearly nullify each other's effect and the resulting small net change in the missile's trajectory indicate that the guidance law is overracting to the state estimation error. In our example, the robust guidance law is less sensitive to this error than the pro-nav guidance law, and thus its performance degrades less sharply. Nevertheless, by continuing to increase the magnitude of the state estimation error, we can quickly reach a situation where neither guidance algorithm is capable of hitting the target.

Initial Condition System Analysis

An alternative and perhaps more meaningful approach to assessing the interrelationship between estimation filtering errors and guidance algorithm performance is to determine the impact of estimation errors upon the set of initial conditions from which the desired target is ultimately reachable. To demonstrate this approach, we considered the system given in Figure 1 and assumed a ten second flight time. We assumed that the target had a radius of 2 units (thus we interpreted miss distances of 2 units or less as a hit) and assumed that the vehicle was equipped with enough fuel to provide six seconds of maximum acceleration. Within this framework we used a combination of analytical techniques and computer simulations to determine, for each guidance algorithm and for a variety of estimation filtering error conditions, sets of initial conditions which resulted in a successful target intercept.

The overall results of our findings are shown in Figure 12. The area between each pair of lines represents the sets of initial conditions from which our vehicle can reach the target in each of the following five situations: 1) Using the fuel optimal guidance law with no state estimation errors 2) Using the pro-nav guidance law with no state estimation errors 3) Using the robust guidance law with no state estimation errors 4) Using the pro-nav guidance law with five percent state estimation errors 5) Using the robust guidance law with five percent state estimation errors. Due to the large scale involved the results may be difficult to interpret.

To facilitate interpretation we displayed the fourth quadrant of Figure 12 separately in Figure 13. For the associated situation, each of these lines represents the the boundary of the set of initial conditions from which the missile can hit the target. From this diagram we can see that even with perfect state information neither the pro-nav nor the robust guidance law can match the performance of the fuel optimal guidance law. For instance, if the vehicle and target have zero initial relative crossrange velocity then the fuel optimal guidance law exhibits a crossrange intercept capability of 4,130 units. For the robust guidance law and the pro-nav guidance law these values are 4,030 units and 4,000 units respectively. These differences are relatively small with the robust guidance law having only 2.42 percent less crossrange position capability given a zero initial crossrange velocity than the fuel optimal guidance law. The pro-nav guidance law was slightly worse with 3.15 percent less capability under these same conditions. However, it should be noted that these are best case results and that the crossrange intercept performance gaps widen as the magnitude of the initial crossrange velocity increases. For instance, if we assume that the vehicle has an initial y-velocity of -200 units/sec relative to the target then the fuel optimal guidance law exhibits a crossrange intercept capability of 2,130 units. For the robust guidance law and the pro-nav guidance law these values are 2,035 units and 2,000 units respectively. Thus, for this nonzero value of initial crossrange velocity the performance gaps have increased to 4.46 percent less crossrange position capability for the robust guidance law than the fuel optimal guidance law and 6.10 percent less crossrange position capability for the pro-nav guidance law than the fuel optimal guidance law.

From Figure 13 it is clear that the performance of both the pro-nav and the robust guidance algorithm are both greatly affected by the presence of five percent state

estimation errors. In this case the robust guidance law now has 15.25 percent less crossrange position capability given zero initial crossrange velocity than the fuel optimal guidance law. The pro-nav guidance law fares even worse with 24.94 percent less crossrange position capability under these same conditions. Similar to the results presented with no state estimation errors, these are best case results and the crossrange intercept performance gaps widen as the magnitude of the initial crossrange velocity increases.

Comparison of Results

In both of the analyses presented above we saw that the performance degradation of the pro-nav and robust guidance laws due to state estimation error is very significant when compared to the performance gaps between the pro-nav and robust guidance law and the fuel optimal guidance law with no state estimation error. We discounted the results of any specific single run analysis because the relative performance of the guidance algorithms can differ depending on the specifics of the engagement condition. However, we feel that the initial condition system analysis presents a relatively clear indication of performance. From this analysis, note the three nearly coincident lines in Figure 13, we can see that relatively little performance (from the theoretical optimum) is lost in implementing either the pro-nav or robust guidance algorithm with perfect state information. Conversely, a large performance gap exists when either algorithm is implemented with a small level of state estimtion error.

The main intent for including the robust guidance algorithm was not to advertise it as a replacement for pro-nav guidance. Instead, it was included to demonstrate that it is indeed possible to improve upon the performance of pro-nav guidance in the presence of state estimation errors. By illustrating the size of the performance gap and the realistic possibility of closing this gap, we hope to motivate further research along these lines. We feel that the key to this and future improvement lies in exploiting the interrelationship between estimation and guidance by explicitly incorporating estimation errors into the robust guidance algorithm.

CONCLUSIONS AND RECOMMENDATIONS

In this paper we have attempted to demonstrate the significance of state estimation errors on guidance algorithm performance. This was accomplished by comparing the performance of pro-nav and robust guidance to ideal fuel optimal guidance both with and without the presence of state estimation errors. Even with perfect state information neither guidance algorithm could achieve the theoretical optimum performance. However, the magnitude of their suboptimality provided a good frame of reference to assess the magnitude of performance degradation due to imperfect state information.

From our studies it is clear that imperfect state estimates can severely degrade guidance law performance. At the same time, the performance of the robust guidance law demonstrates that it may indeed be possible to improve upon the robust performance characteristics of pro-nav guidance. We feel that future progress in solving this problem lies with a better understanding of the interrelationship between the state estimator and the guidance algorithm. This area is currently under further research.

REFERENCES

Athans, M. and P.L. Falb, Optimal Control, McGraw-Hill, New York, 1966.

Bryson, A.E. and Y.C. Ho, Applied Optimal Control, Halsted Press, New York, 1968.

Ting, T., 'On Developing a Robust Missile Guidance Algorithm,' Proceedings of the 1992 American Control Conference, Chicago, IL, June 1992, pp. 2380-2385.

Zarchan, P., Tactical and Strategic Missile Guidance, AIAA, Washington D.C., 1990.

1992 ACC/TP7

On Developing a Robust Missile Guidance Algorithm

Thomas L. Ting

Honeywell Systems and Research Center
3660 Technology Drive
Minneapolis, MN 55418

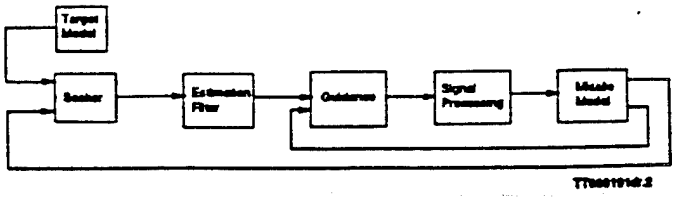


Figure 1: Typical Missile Divert Control System

ABSTRACT

In this paper we consider the problem of exoatmospheric missile/target engagements with imperfect state information. We present a newly developed robust guidance algorithm which achieves better hit performance and less fuel consumption than a proportional navigation (pro-nav) guidance algorithm for certain engagement scenarios. These preliminary results lead us to believe that robust guidance algorithms can eventually be developed which will globally outperform existing guidance algorithms in dealing with uncertainties.

INTRODUCTION

This paper describes the preliminary development of a robust guidance algorithm for use in exoatmospheric missile/target engagement problems. The primary motivation for our guidance research was twofold. First, we wanted to develop an algorithm which was straightforward to implement; i.e. a minimum of ad-hoc fixes and dependence on designer skill. Second, we wanted to improve upon the performance of a popular existing algorithm, the proportional navigation (pro-nav) guidance algorithm, both in terms of fuel consumption and hit probabilities. Much of this effort was directed towards improving upon the robustness of pro-nav guidance algorithms, that is improving the performance in the face of uncertainties. In our particular study these uncertainties were characterised by errors in the state estimates used by the guidance algorithms.

BACKGROUND AND MOTIVATION

The linear and rotational motions of a exoatmospheric missile are typically controlled by firing thrusters. The most elementary thrusters possess an ON/OFF characteristic. More complicated thrusters are throttleable, featuring a variable thrust level. The attitude and divert control thrusters are often subdivided into separate groups with each group of thrusters controlled by separate attitude and divert control systems. These systems are responsible for processing feedback information about the current states of the missile relative to its desired target and issuing appropriate firing commands. For guidance purposes, the divert control system is crucial because it is responsible for controlling the non-rotational motion of the missile.

A block diagram of a typical missile divert control system is shown in Figure 1, and its operation can be summarised as follows. At the beginning of each guidance cycle, values of the angular rates of the vehicle and filter estimates of the missile's linear position, velocity and acceleration relative to the target model are fed back into a guidance algorithm. This algorithm processes this information and generates a vector of desired linear acceleration commands. These linear acceleration commands are then transformed into individual divert thruster commands which are applied to each of the divert thrusters.

If we employ a standard pro-nav guidance algorithm then the divert control system design is complicated by the presence of ON/OFF divert control thrusters. Pro-nav guidance algorithms are usually allowed to output a continuous valued vector of linear acceleration commands. This results in a continuous set of divert thruster commands which are incompatible with ON/OFF divert thrusters. This complication may be addressed by implementing a scheme, such as pulse-width modulation (PWM), which converts a continuous valued input vector of divert thruster commands into an ON/OFF valued output vector of divert thruster commands.

Proportional navigation guidance also does not address the issue of minimising divert thruster firings. In our past experiences pro-nav guidance has on occasion exhibited an excessive amount of divert thruster firing activity. This phenomenon can be treated by appending a deadzone filter to the divert control system. Such filters process the guidance algorithm's output vector of linear acceleration commands prior to their transformation into individual divert thruster commands. The size of the 'optimal' deadzone is typically determined experimentally. Although this solution generally produces satisfactory results, we are not satisfied with the absence of a formal design procedure for determining the size of the deadzone.

Our objective in this paper is to describe the preliminary development of a robust guidance algorithm which

accomplishes four main goals in comparison with a pronav guidance scheme. First, it produces discrete valued linear acceleration commands which are completely compatible with ON/OFF divert thrusters. Second, it avoids the use of deadzones or determines them in an analytic fashion. Third, it improves hit performance by making the terminal miss distance more robust to filter estimation errors. Finally, it reduces overall divert fuel consumption.

A PROPOSED NEW ROBUST GUIDANCE ALGORITHM

Main Ideas

We shall first explain the basic principles behind our guidance algorithm. The technical details will be illustrated using a generic two dimensional guidance problem later in this section.

The development of our guidance algorithm consists of three main steps; 1) A miss distance calculation 2) A miss distance radius due to filter estimate errors and 3) A miss distance deadzone concept. At any point in a missile/target simulated engagement we can compute the Zero Effort Miss distance, ZEM. This represents the distance by which the missile will miss the target assuming that the missile acceleration is zero, that is no external effort (force) is acting upon the missile, for the duration of its flight. For our guidance algorithm, we are interested in not only the value of ZEM, but also of the sensitivity of ZEM with respect to our state estimates of the relative linear components of acceleration, velocity and displacement. These sensitivities are useful because they represent the incremental change in ZEM which will accompany a unit incremental change in each state.

As stated earlier, one of our foremost design objectives is robustness to state estimation errors. Thus, in addition to computing a single miss distance value ZEM, we compute a miss distance radius R due to possible state estimation errors. This radius R is computed as follows. First, we determine a maximum magnitude filtering error bound for each of the relative acceleration, velocity and displacement states. Then, using the sensitivities of ZEM with respect to these states, we compute the maximum magnitude error in ZEM associated with the current values of the states. This maximum magnitude error is the miss distance radius R.

The miss distance deadzone concept is very simple. It determines the output of the guidance algorithm in the form of a velocity correction. If the absolute value of the computed miss distance ZEM is sufficiently small then the velocity correction is set to zero. However, if the absolute value of ZEM is large then a velocity correction is issued

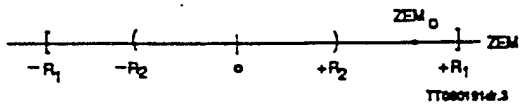


Figure 2: Miss Distance Deadzone Concept

which should correct the ZEM to zero. In our algorithm the boundary between large and small is given by the miss distance radius R. If the absolute value of ZEM is smaller than R then the guidance algorithm outputs a velocity correction of zero. However, if the absolute value of ZEM is greater than R then the guidance algorithm outputs a velocity correction which should correct the ZEM to zero. Analytically, this is done by outputting a velocity correction which is equal to the current ZEM divided by the sensitivity of the ZEM with respect to the velocity state variable under consideration.

In this paper we restrict our attention to outputting only y-component velocity corrections. In this context, the miss distance deadzone concept can be interpreted as shown in Figure 2. Here, the values along the horizontal axis represent various values of ZEM. Suppose the computed value of ZEM is given by ZEM_o. If the miss distance radius R is given by R_1 then the y-velocity component correction is set to zero. However, if the miss distance radius R is given by R_2 then the y-velocity component correction is set to $\frac{ZEM}{\frac{\partial ZEM}{\partial v_y}}$.

Algorithm Details

We will now illustrate the details of our guidance algorithm using the generic two dimensional guidance problem shown in Figure 3. Here, the target and the missile are represented by point mass models where the target is denoted by the letter T and the missile is denoted by the letter M. The relative positions of the missile and target are represented with respect to a xy-coordinate frame whose origin is always coincident with the location of the target. In this coordinate frame the target has zero velocity and acceleration and is always located at the origin. Conversely, the missile always begins with a negative x-component and is represented by a relative distance from the target (origin). The missile is also moving relative to the target (origin) with some velocity and acceleration. To further simplify our problem, we assumed no gravity effects (i.e. an exoatmospheric engagement) and that the target maintained a constant acceleration.

Our guidance algorithm begins with a computation of the Zero Effort Miss. This miss distance can be defined in several different ways. For our two dimensional problem we chose to define the ZEM as the distance between the target and the missile when the target crosses the line x=0 (i.e. the y-component of the trajectory when

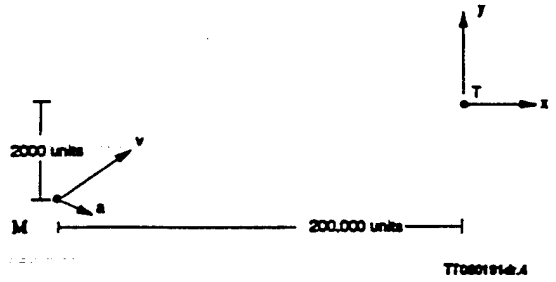


Figure 3: Two Dimensional Guidance Problem

it crosses the y-axis).

At any time t_o let us denote the values of the relative x and y components of accelerations, velocities and distances of the missile and target by $a_{xo}, a_{yo}, v_{xo}, v_{yo}, x_o$ and y_o. Then assuming constant target accelerations and zero missile accelerations, the future values of relative x and y components of accelerations, velocities and distances of the missile and target at any future time $t_o + t_1$ are given by

$$a_x(t_o + t_1) = a_{xo} \quad (1)$$

$$a_y(t_o + t_1) = a_{yo} \quad (2)$$

$$v_x(t_o + t_1) = v_{xo} + \int_{t_o}^{t_o+t_1} a_x(t)dt = v_{xo} + a_{xo}t_1 \quad (3)$$

$$v_y(t_o + t) = v_{yo} + \int_{t_o}^{t_o+t_1} a_y(t)dt = v_{yo} + a_{yo}t_1 \quad (4)$$

$$x(t_o + t) = x_o + \int_{t_o}^{t_o+t_1} v_x(t)dt = x_o + v_{xo}t_1 + .5a_{xo}t_1^2 \quad (5)$$

$$y(t_o + t) = y_o + \int_{t_o}^{t_o+t_1} v_y(t)dt = y_o + v_{yo}t_1 + .5a_{yo}t_1^2. \quad (6)$$

From these expressions we may compute the time to go t_{go}. This represents the length of time before the missile crosses the y-axis. At any time t_o this value is given by solving the equation

$$x(t_o + t_{go}) = x_o + v_{xo}t_{go} + .5a_{xo}t_{go}^2 = 0 \quad (7)$$

for t_{go}. Two separate cases exist. If $a_{xo} \geq 0$, that is if the target is accelerating towards the missile, then

$$t_{go} = \frac{-v_{xo}}{a_{xo}} + \sqrt{\frac{v_{xo}^2}{a_{xo}^2} - \frac{2x_o}{a_{xo}}}. \quad (8)$$

Alternatively, if $a_{xo} \leq 0$, that is if the target is accelerating away from the missile, then

$$t_{go} = \frac{-v_{xo}}{a_{xo}} - \sqrt{\frac{v_{xo}^2}{a_{xo}^2} - \frac{2x_o}{a_{xo}}}. \quad (9)$$

Once t_{go} has been computed the miss distance ZEM is given by the value of y at t_{go} or

$$ZEM = y(t_{go}) = y_o + v_{yo}t_{go} + .5a_{yo}t_{go}^2. \quad (10)$$

In addition to the miss distance ZEM we are also interested in computing the sensitivities of ZEM with respect to the relative x and y components of acceleration, velocity and displacement. Using equations 8-10 these sensitivities can be computed in a straightforward fashion to yield

$$\frac{\Delta ZEM}{\Delta a_{yo}} = \frac{dZEM}{da_{yo}} = .5t_{go}^2 \quad (11)$$

$$\frac{\Delta ZEM}{\Delta v_{yo}} = \frac{dZEM}{dv_{yo}} = t_{go} \quad (12)$$

$$\frac{\Delta ZEM}{\Delta y_o} = \frac{dZEM}{dy_o} = 1 \quad (13)$$

$$\frac{\Delta ZEM}{\Delta a_{xo}} = \frac{dZEM}{da_{xo}} = \frac{(v_{yo} + a_{yo}t_{go})(x_o + v_{xo}t_{go})}{a_{xo}(a_{xo}t_{go} + v_{xo})} \quad (14)$$

$$\frac{\Delta ZEM}{\Delta v_{xo}} = \frac{dZEM}{dv_{xo}} = \frac{-(v_{yo} + a_{yo}t_{go})t_{go}}{a_{xo}(a_{xo}t_{go} + v_{xo})} \quad (15)$$

$$\frac{\Delta ZEM}{\Delta x_o} = \frac{dZEM}{dx_o} = \frac{-(v_{yo} + a_{yo}t_{go})}{(a_{xo}t_{go} + v_{xo})}. \quad (16)$$

Let us denote our six state estimates by $\hat{x}, \hat{y}, \hat{v}_x, \hat{v}_y, \hat{a}_x$ and $\hat{a}_y$. We must first determine the magnitude of possible estimation errors associated with each of these six state variables. We denote these error magnitudes by $\Delta x, \Delta y, \Delta v_x, \Delta v_y, \Delta a_x$ and Δa_y. In a true design problem these values could be determined iteratively by beginning with initial guesses with future updating based on actual simulation results. For our problem we simply set the magnitude of each estimation error equal to a fixed percentage of the corresponding state estimate.

From the state estimation error bound magnitudes and the sensitivities given in Equations 11-16, the miss distance radius R may be computed as

$$R = \| \frac{\Delta ZEM}{\Delta \hat{x}} \Delta \hat{x} \| + \| \frac{\Delta ZEM}{\Delta \hat{y}} \Delta \hat{y} \| + \| \frac{\Delta ZEM}{\Delta \hat{v}_x} \Delta \hat{v}_x \| + \| \frac{\Delta ZEM}{\Delta \hat{v}_y} \Delta \hat{v}_y \| + \| \frac{\Delta ZEM}{\Delta \hat{a}_x} \Delta \hat{a}_x \| + \| \frac{\Delta ZEM}{\Delta \hat{a}_y} \Delta \hat{a}_y \|. \quad (17)$$

R represents the maximum possible uncertainty in the magnitude of the ZEM due to state estimation errors.

The output of our guidance algorithm is an incremental velocity change command, δv_y. As stated earlier, we use a very elementary miss distance deadzone concept. If the computed miss distance is less than R, then δv_y is

set to zero. If not, then δv_y is commanded such that it corrects the next computed miss distance to zero. Mathematically, this is expressed as

$$\delta v_y = \left\{ \begin{array}{c} 0 \; if \; \| \; ZEM \; \| \leq \mathrm{R} \\ \frac{ZEM}{\frac{\Delta ZEM}{\Delta v_y}} \; if \; \| \; ZEM \; \| > \mathrm{R} \end{array} \right\}. \quad (18)$$

A major difference between our guidance algorithm and pro-nav guidance is the form of the commanded output. Pro-nav guidance algorithms usually output linear acceleration commands. Our guidance algorithm ouputs linear velocity incremental change commands. That is, instead of commanding acceleration in each linear direction, our guidance algorithm issues desired incremental velocity changes in each linear direction.

As described earlier, pro-nav guidance outputs are often passed through deadsone filters which zero out all input which fall below a specified threshold. In contrast to previous ad-hoc methods, our guidance algorithm was designed with an automated procedure to handle the selection of an appropriate deadsone threshold. This threshold was selected such that our guidance algorithm output incremental velocity commands which were entirely compatible with ON/OFF thruster characteristics. This procedure operated as follows. Given the mass of the missile and the guidance cycle rate we could compute the maximum linear velocity change, Δv_{max}, achievable during a single guidance cycle. This value was then selected to be the threshold of our velocity deadsone filter. Thus, our true guidance algorithm output is computed as follows:

$$\delta v_y = \left\{ \begin{array}{c} 0 \; if \; \| \; \delta v_y \; \| < \Delta v_{max} \\ \Delta v_{max} \; if \; \delta v_y \geq \Delta v_{max} \\ -\Delta v_{max} \; if \; \delta v_y \leq -\Delta v_{max} \end{array} \right\}. \quad (19)$$

After preliminary testing with our guidance algorithm some additional features were added to enhance performance. First, the miss distance deadsone was shutoff for the last second of the engagement. Whenever t_{go} was one second or less the velocity correction was computed according to the second expression of Equation 18 without regard to the inequality conditions. Second, the miss distance radius R given by Equation 17 was subject to upper and lower bounds. The upper bound was appended because we wanted to ensure that when t_{go} reached one second, ZEM was sufficiently small so that the missile could successfully intercept the target. The lower bound was appended to ensure that R did not become too small too quickly which could result in a ZEM response which oscillates about zero.

The upper and lower bounds on R were set equal to a constant multiple of t_{go}. The gain on the upper bound was determined by the relative acceleration capabilities of the missile and the target. For instance, since we know

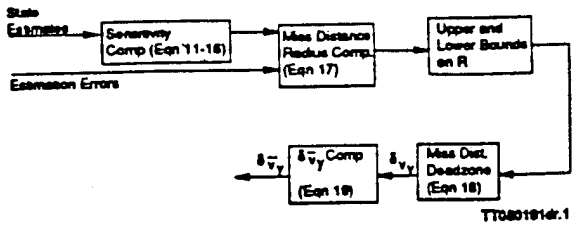


Figure 4: Block Diagram of Our New Guidance Algorithm

that the miss distance deadsone is shutoff when $t_{go} = 1$ second, we want ZEM to be small enough such that it can be reduced to zero in one second or less. The maximum ZEM correction achievable in time t is given by

$$\Delta ZEM = .5max(a_{ymissile} - a_{yo})t^2. \quad (20)$$

In our example ΔZEM was approximately $30t^2$. Thus, we want the ZEM to be less than 30 when $t_{go} = 1$ second. Consequently, we set the upper bound on R to be $30t_{go}$.

The lower bound on R was not computed analytically. We simply experimented with various gains unitl we found one we deemed satisfactory. For our example we settled on a lower bound of $10t_{go}$.

A block diagram illustrating the full computations of our new guidance algorithm is shown in Figure 4.

SIMULATION RESULTS

We applied our new guidance algorithm to a two-dimensional simulated engagement between a missile and a target. The initial conditions of this simulation are characterised as follows. First, the missile and target began 200,000 units apart in the x-direction and 2,000 units apart in the y-direction. Second, the missile began with an initial relative closing velocity of 20,000 units/sec in the x-direction and 400 units/sec in the y-direction. Finally, the missile began with an initial relative acceleration of 50 units/sec^2 in the x-direction and -30 units/sec^2 in the y-direction. The initial relative acceleration in the y-direction was derived from the fact that the target was assumed to have a constant acceleration of 30 units/sec^2 in the y-direction while the missile was given zero initial acceleration in this direction.

We ran simulations of the engagement described above using three different guidance algorithms. These algorithms were standard proportional navigation, standard proportional navigation with a deadsone filter, and an algorithm based on our new guidance scheme described earlier. The size of the deadsone augmenting the standard

proportional navigation algorithm was selected by trial and error in an attempt to achieve a compromise between good hit performance and minimal divert fuel usage.

The different guidance algorithms were compared on their ability to perform in the scenario described above. Performance was characterised by two main factors: terminal miss distance and divert thruster ontime. These measures were chosen because they address two main questions regarding any missile engagement: 1) Did the missile hit the target? and 2) How much fuel do we need to put in the missile so that it can hit the target?

For comparison purposes we conducted two sets of simulation runs. The first set of runs were conducted assuming no state estimation errors. The second set of runs were conducted assuming a ten percent estimation error in the relative x-distance between the missile and target. We could have conducted simulation runs assuming estimation errors in more than one state, but we felt that the two sets of runs described above were sufficient to illustrate our results.

The results of the first set of simulation runs (each of the three guidance schemes run separately with no state estimation errors) are shown in Figure 5. The simulation results obtained using a proportional navigation guidance algorithm are shown by the solid black lines. By appending a deadzone to the proportional navigation algorithm we obtained the simulation results shown by the small dotted lines. Finally, the simulation results obtained using the guidance algorithm described in this paper are shown by the medium dotted lines.

It is clear that with no state estimation errors, all three guidance algorithms achieve comparable miss distance performance. In fact, the pro-nav guidance and the pro-nav guidance with the added deadzone achieve nearly identical miss distances. Our guidance algorithm results in a slightly larger terminal miss distance, but this difference is almost negligible when compared to the size of most targets.

Conversely, the three guidance schemes differ greatly in the amount of divert thruster ontime. This is significant because with ON/OFF divert thrusters the amount of divert thruster fuel consumed is directly proportional to the amount of thruster ontime. Since additional fuel increases the weight of the missile which must be carried into space, it is desirable to minimise the fuel requirements of the missile.

From Figure 5 it is evident that the pro-nav guidance algorithm results in the largest divert thruster ontime (divert fuel consumption). This value is reduced (in our example) by approximately fifteen percent by appending a deadzone to the pro-nav guidance algorithm. However, by implementing the our guidance algorithm it is possible

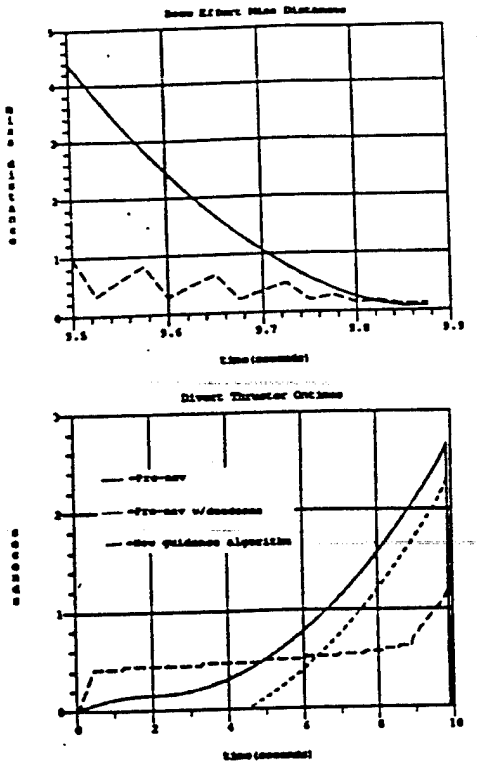


Figure 5: Simulation Results With No State Estimation Errors

to reduce divert thruster ontime by more than fifty-five percent as compared to a standard pro-nav guidance algorithm. Thus, in this case our guidance algorithm can greatly reduce the amount of divert thruster activity with a very minimal impact upon miss distance performance. In fact, this inpact is so small that it probably would not change the likelihood of hitting a target.

The results of the second set of simulation runs (each of the three guidance schemes run separately with a ten percent state estimation error in the relative x-distance between the missile and target) are shown in Figure 6. The results corresponding to the three different guidance algorithms are indicated by the same type of lines described for Figure 5.

In this example the pro-nav guidance algorithm and the pro-nav guidance with the appended deadzone again achieve nearly identical miss distance performance. Unfortunately, both guidance algorithms result in a relatively large terminal miss distance. The deterioration in miss distance performance as compared to the simulation with no state estimation errors indicates that neither of these two algorithms is particularly robust to state estimation errors. By contrast, our guidance algorithm achieves very good miss distance performance in spite of this state estimation error. A comparison of Figures 5 and 6 indicates that the terminal miss distance performance

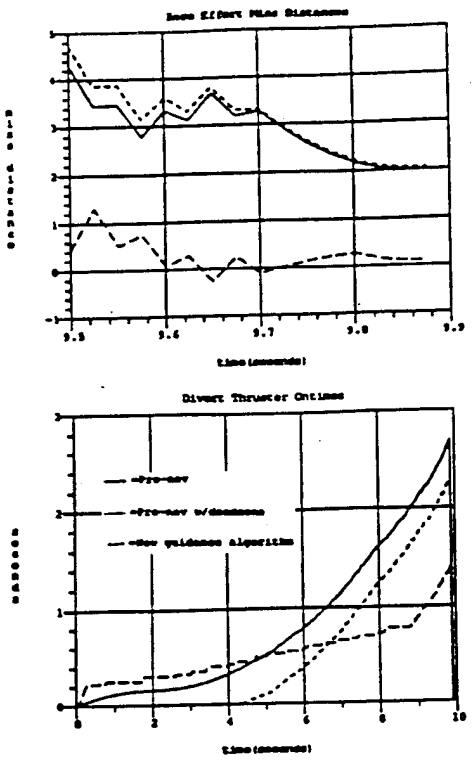


Figure 6: Simulation Results With Ten Percent X-State Estimation Errors

of our guidance algorithm deteriorates very little in the presence of this state estimation error.

It is also seen that the presence of the state estimation did not affect the relative fuel consumption characteristic of the three guidance schemes. The pro-nav guidance algorithm again required the largest fuel consumption, followed by the pro-nav guidance with the appended deadsone and finally our new guidance algorithm. As before, appending the deadsone to the pro-nav guidance algorithm reduced divert thruster ontime by roughly fifteen percent, while our new guidance algorithm reduced divert thruster ontime to roughly one half of the value associated with the pro-nav guidance algorithm. However, in this case the new guidance algorithm not only achieved greatly reduced fuel consumption in comparison with pro-nav guidance, but also achieved a significant advantage in terminal miss distance performance.

In the transition from the perfect information case to the case with state estimation error, we note that the fuel consumption associated with all three guidance algorithms increased. This is expected because the presence of state estimation errors leads to erroneous thruster commands and subsequent thruster firings. In such situations, fuel is wasted not only on these erroneous firings themselves, but in the firings which must be made to correct the harmful effects caused by the erroneous firings. We expect that on average as the magnitude of the state estimation errors increase, the percentage increase in fuel consumption over the perfect information case will also increase.

CONCLUSIONS AND RECOMMENDATIONS

We have explored some new ideas for developing a robust guidance algorithm. These ideas were incorporated into a first cut version of a new guidance algorithm. This algorithm performed favorably in a simulated missile/target engagement in comparison with two versions of standard pro-nav guidance. However, we must caution that these results are engagement dependent and favorable results are not always obtained. In addition, although we were able to avert a majority of the ad-hoc procedures and ON/OFF thruster complications associated with the pro-nav guidance algorithm, we were not able to remove all of the ad-hoc procedures from our guidance algorithm design, i.e. the selection of the lower bound on R. Further research is needed to better understand the full benefits and drawbacks of our ideas. It is our hope that this paper may help to focus attention on this problem and stimulate research into the development of more robust guidance algorithms.

ACKNOWLEDGEMENTS

The author wishes to thank Dr. James Krause of Honeywell SRC for his technical consultation. This work was conducted under Honeywell SRC internal research and development funding.

REFERENCE

Zarchan, P., "Tactical and Strategic Missile Guidance," Washington D.C., AIAA, 1990.

1993 CDC

Fault Detection in the Presence of Modeling Uncertainty *

Pramod P. Khargonekar † and Thomas L. Ting §

ABSTRACT

In this paper we describe a new approach for detecting faults in systems in the presence of modeling uncertainty. Our approach is interactive and relies on processing true time domain system measurement data to determine whether or not the system is operating within an expected range of behaviors. Our results are applicable to either open or closed loop systems and can be implemented in a numerically efficient fashion.

INTRODUCTION

As today's state-of-the-art control systems (i.e. military, aerospace, chemical processes etc.) become increasingly complex the problem of fault detection is gaining in importance. For many of these systems the presence of a single undetected fault can lead to greatly reduced performance or worse yet a catastrophic failure. Fault detection algorithms attempt to determine when a system is operating outside of its range of expected behaviors. Once a fault is detected various means exist to isolate the fault and allow the system to operate at a suboptimal level. Some control designs are sophisticated enough to achieve optimal reconfiguration in the presence of a fault.

Modeling uncertainty and noise inputs both complicate the problem of fault detection. For example, suppose a control engineer possessed a perfectly accurate mathematical model of a system and suppose all the inputs could be measured. Then the fault detection problem may be solved through the following basic principle. If the measured relationship between any two signals within the system does not agree with the relationship predicted by the model then there must exist a fault in the system.

Now suppose this same control engineer has a system with either some unmeasurable noise inputs and/or some level of modeling uncertainty. Due to the presence of these uncertainties, it is clear that the principle described above is no longer directly applicable to this problem. In this case the job facing the fault detection algorithm can be broken into two steps. First, it must determine whether a discrepancy exists between the measured relationship between any two signals within the system and the expected relationship predicted by the model. Second, if a discrepancy does exist then it must determine whether this information is truly indicative of a fault in the system, or is simply the result of noise or modeling uncertainty. Traditionally, fault detection algorithms have been analyzed in a probablilistic setting to account for random noise [2,3,4,6,10]. This probabilistic approach is, however, difficult to use in the presence of unmodeled dynamics which is typically described in a deterministic setting. Recently, fault detection problems in the presence of unmodeled dynamics have been investigated [1,7,8,9].

In this paper we investigate a fault detection problem featuring significant modeling uncertainty due to unmodeled dynamics. A typical example is unmodeled high frequency dynamics. We develop a fault detection algorithm that accounts for these unmodeled dynamics. Our approach is closely related to the recent work on robust identification as well as model validation [5,7].

MATHEMATICAL PRELIMINARIES

There exist a variety of mathematical concepts to characterize the time and frequency domain behavior of plants and their associated input/output signals. In this section we briefly review those concepts which are pertinent to our discussion.

Signal and Operator Norms

Let $\mathcal{L}_2{}^n$ be the set of all square integrable functions from $[0,\infty)$ to $\mathcal{R}^n$

$$\mathcal{L}_2^n = \{u(t) : \int_0^\infty u^T(t)u(t)dt < \infty\}. \qquad (1)$$

For all signals $u(t)$ in this set we may define the $\mathcal{L}_2$ norm

$$\|u\|_2 = \sqrt{\int_0^\infty u^T(t)u(t)dt}. \qquad (2)$$

*Supported in part by the Airforce Office of Scientific Research under contract no. AFOSR-F49620-92-C-0056

† Dept. of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109-2122 Tel. No. (313) 764-4328, Fax No. (313) 763-1503

§ Honeywell Systems and Research Center, 3660 Technology Drive, Minneapolis, MN 55418, Tel. No. (612) 951-7299, Fax No. (612) 951-7438.

This norm can be interpreted as the "energy" in the signal $u(t)$. For a square integrable function u from $(0,T)$ to $\mathcal{R}^n$, define the partial $\mathcal{L}_2$ norm as

$$\|u\|_2^{(0,T)} = \sqrt{\int_0^T u^T(t)u(t)dt}. \quad (3)$$

This norm represents the "energy" in the signal $u(t)$ over the time interval $(0,T)$. To emphasize the most recent signal information define a weighted partial $\mathcal{L}_2$ norm as

$$\|u\|_{2\sigma}^{(0,T)} = \sqrt{\int_0^T e^{\sigma(t-T)}u^T(t)u(t)dt}. \quad (4)$$

In this weighted norm all the past values of u are weighted by a factor which decays exponentially as a function of time. This limits the ability of accumulated past signal values to dilute the effects of new information. The rate of decay is controlled by the selection of the constant σ.

Modeling Uncertainty

Mathematical models are an attempt to analytically represent the behavior of a true physical system. Control designers rely heavily on such models in designing modern, high performance controllers. In reality, however, true physical systems are never perfectly represented by a mathematical model. Thus, it is important for a control engineer to design controllers which provide not only good nominal performance but are also robust, i.e. which perform well in spite of plant uncertainties.

One popular approach to synthesizing robust controllers is to build uncertainty into the plant model. In such cases, the plant is not represented by a single nominal plant model, P_o(s), but is instead allowed to be any element in a family of plant models. One typical family of plant models is given by

$$\mathcal{F} = \{(I + \Delta W(s))P_o(s) : \|\Delta\|_\infty \leq \delta_{max}\} \quad (5)$$

where P_o(s) is the nominal plant model, Δ represents an unknown, linear, time-invariant, stable, proper, rational, norm-bounded perturbation, and W(s) is a frequency dependent weighting function which characterizes the relative magnitude of modeling uncertainty at various frequencies. Typically, W(s) takes on larger values at higher frequencies reflecting the inability to model accurately at these frequencies. Here the bound, δ_{max}, on Δ is the $\mathcal{L}_2$-induced operator norm which represents the maximum amount of modeling uncertainty associated with $\mathcal{F}$.

A KEY BOUND

In this section we describe a key time-domain bound which forms the basis of our fault detection algorithm. Consider the generic system shown in Figure 1. Here, the input signal u and the output signal y are related by the transfer function $\Delta(s)$ through the expression $y = \Delta(s)u$.

Suppose that the magnitude portion of the frequency response of $\Delta(s)$ has a frequency weighted bound. In other words, suppose there exists a nonnegative real function $\delta(\omega)$ such that $\|\Delta(j\omega)\|_\infty \leq \delta(\omega)$ for all frequencies ω. Moreover, suppose that $\delta(\omega)$ is such that there exists a rational, minimum phase, proper, stable function $W(s)$ such that $|W(j\omega)| = \delta(\omega)$ for all frequencies ω. Then the system shown in Figure 1 can be redrawn as shown in Figure 2 with $\Delta = \hat{\Delta}W$ where $\|\hat{\Delta}\|_\infty \leq 1$.

Since $\| \hat{\Delta} \|_\infty \leq 1$ then by definition

$$\| y \|_2^{2(0,t)} \leq \| v \|_2^{2(0,t)} \quad (6)$$

for any arbitrary time t. This inequality can be interpreted as follows. For any time t the energy of the output of $\hat{\Delta}$ over the interval (0, t) is smaller than the energy of the input of $\hat{\Delta}$ over this same interval.

More generally, suppose two signals v and y are related by a causal operator $\hat{\Delta}$ such that $\|\hat{\Delta}\| \leq 1$ where the norm on $\hat{\Delta}$ is the $\mathcal{L}_2$induced norm. Then it is easy to see that

$$\| y \|_2^{2(0,t)} \leq \| v \|_2^{2(0,t)} \quad (7)$$

holds. Indeed, it has been shown in [7] that if $\hat{\Delta}$ is allowed to be linear time-varying, then this is a tight bound.

In real time fault detection algorithms, it is desirable to discount past information. For instance, suppose $\hat{\Delta}(s)$ is a linear, time-invariant, proper, rational, stable function with no poles in $Re(s) \geq -\sigma$. Moreover, suppose

$$\|\hat{\Delta}\|_\sigma := \sup_{Re(s)\geq -\sigma} \sigma_{max}(\hat{\Delta}(s)) \leq 1. \quad (8)$$

Then it is possible to show using the weighted partial norm that

$$\| y(\tau) \|_{2\sigma}^{2(0,t)} \leq \| v(\tau) \|_{2\sigma}^{2(0,t)} \quad (9)$$

for any arbitrary time t.

To facilitate implementation of either Equation (??) or (??) in real-time fault detection algorithms, it is imperative to minimize computational and memory storage requirements associated with finding the upper bounds. For Equation (??) this is accomplished by defining

$$\hat{V}(t) = \| v(\tau) \|_{2\sigma}^{2(0,t)} = \int_o^t e^{\sigma(t-T)}v(\tau)^T v(\tau)d\tau. \quad (10)$$

Here $\hat{V}$(t) is a monotonically increasing function of t and its derivative is given by

$$\frac{d\hat{V}}{dt}(\tau) = -\sigma\hat{V}(\tau) + v(\tau)^T v(\tau). \quad (11)$$

Now $\hat{V}$(t+Δt) can be updated recursively by numerically integrating Equation (??). This computation requires only the most recent measurement value $v(t)$, the most recent computed value of $\hat{V}(t)$, the timestep Δt, and the constant σ. In addition to computational simplicity this approach reduces memory requirements by eliminating the need to store all values of past measurements from time zero to the present. The upper bound for Equation (??) can be obtained by following a similar procedure.

FAULT DETECTION ALGORITHM

In this section we use the key bound described in the previous section to develop our fault detection algorithm. We first describe the general ideas behind our algorithm and outline their implementation. Next, we address more specific algorithm implementation issues such as selecting an appropriate fault detection threshold to account for noise, modeling uncertainty and the tradeoff between various error probabilities.

Algorithm Outline

We describe our algorithm in the context of a typical open loop system as shown in Figure 3. This system contains a plant P, featuring multiplicative modeling uncertainty, which is allowed to be any member of a family of plants $\mathcal{F}$ described by Equation (??). For simplicity we selected a weighting function $W(s)$ such that δ_{max} is normalized to one. In addition to modeling uncertainty, the plant P also features an additive failure input signal f. This signal is zero under normal plant operations and is nonzero when the plant is operating in a failed state. Each possible failure of the plant is associated with a different failure signal f. For example, if a system is subject to three different types of failures, then there would be three different failure input signals f_1, f_2, and f_3. The characteristics of each f are intended to replicate the impact an associated failure would have on the plant. It is assumed that the plant operator has complete knowledge of all possible failure signals, but does not know which, if any, signals are present at any given time.

The open loop system features a plant output measurement y corrupted by additive measurement noise n. Although individual values of n are unknown, we assume that certain properties of n are known. For instance, n may be represented by a stochastic noise model such as a Gaussian distribution with zero mean and a fixed standard deviation. Alternatively, n could be represented by a deterministic noise model. Here, n would not be described through probability distributions but would instead be restricted by deterministic constraints, such as weighted $\mathcal{L}_2$or $\mathcal{L}_\infty$norms.

The central idea of our fault detection algorithm is to use available signal measurements to analytically reconstruct the input and output signals to the uncertainty block Δ. This signal set is then examined using the key bound from Equation (??) to determine whether or not it complies with the apriori modeling assumptions on the size of Δ. In particular, we check to ensure that the measurement data could indeed have been generated by a perturbation of size δ_{max} or less. With δ_{max} normalized to 1 our fault detection algorithm basically checks whether or not the output signal, d_{out}, of Δ has more or less energy than the input signal d_{in}. If δ_{max} were not normalized then our fault detection algorithm could simply check whether or not d_{out} has more or less energy than δ_{max} times d_{in}.

From Figure 3, $d_{in} = WP_o r$. This signal is constructable because the reference signal r and the transfer functions W(s) and P_o(s) are all known. Constructing d_{out} is slightly more complicated. From Figure 3 we know that the plant output measurement y is given by

$$y = P_o r + \Delta W P_o r + f + n. \tag{12}$$

Now define $d_{out} = \Delta W P_o r$. Since f and n are unknown we approximate d_{out} as

$$d_{out} \approx y - P_o r = d_{out} + f + n. \tag{13}$$

Assume that the noise signal n is zero. Then using the signals described above we can detect the presence of a fault by computing whether

$$\| y - P_o r \|_{2\sigma}^{2[0,t]} - \| W P_o r \|_{2\sigma}^{2[0,t]} \leq 0 = L_{thresh}. \tag{14}$$

If this inequality holds then d_{in} has more energy than d_{out} in the time interval $[0, t]$. This result is consistent with an uncertainty block Δ of norm less than or equal to one, and thus the fault detection algorithm concludes that no fault is present. Conversely, if this inequality does not hold then by analogous reasoning this result is inconsistent with our modeling assumptions and the fault detection algorithm concludes that a fault must be present.

Operationally, this fault detection algorithm is implemented in three steps. First, any desired r is input to the true physical system and the resulting y is recorded. Second, knowledge about r, P_o, and W, is used to analytically compute $P_o r$ and $WP_o r$. Third, the two norms in Equation (??) are obtained recursively by numerically integrating Equation (??).

A convenient method of checking the inequality in Equation (??) is to display the difference

$$\| y - P_o r \|_{2\sigma}^{2[0,t]} - \| W P_o r \|_{2\sigma}^{2[0,t]} \tag{15}$$

graphically as a function of time. By comparing this value versus L_{thresh} we can easily see whether or not a fault exists.

Thus far we have discussed our fault detection algorithm only in the context of open loop systems. However, the same results are directly applicable to closed loop systems, as shown in Figure 4, provided that the control input u is measurable. If this condition holds then the identical algorithm is applicable to closed loop systems with the sole modification that the reference signal r is replaced by the control input u. Thus, for the closed loop system fault detection algorithm the input and output signals associated with Δ are given by $d_{in} = WP_o u$ and $d_{out} = \Delta W P_o u \approx y - P_o u$. All other aspects of the algorithm remain unchanged. For the remainder of this paper we will continue to discuss our fault detection algorithm in the context of open loop systems noting that all results are also applicable to closed loop systems.

In applications we recommend using the weighted partial $\mathcal{L}_2$-norm instead of the standard partial $\mathcal{L}_2$-norm.

This substitution is allowable because for any plant P in the family $\mathcal{F}$, described by Equation (??), Δ is a stable perturbation which implies that there exists some positive σ such that $\Delta(s)$ has no poles in $Re(s) \geq -\sigma$.

Without the inclusion of the exponential forgetting factor in the partial $\mathcal{L}_2$-norm, it is possible that a long initial period of no fault performance could dilute the ability of a fault associated transient to suddenly alter the difference $\| y - P_o r \|_2^{2[0,t]} - \| WP_o r \|_2^{2[0,t]}$. With the exponential forgetting factor present, the most recent data values are emphasized and this risk is lessened.

Threshold Selection

In reality, the threshold, L_{thresh} (currently zero), on the right hand side of Equation (??), is too stringent for practical applications. This value was derived under the assumption that n is zero. Clearly, the presence of a nonzero n will alter the energy in y, and hence the energy of the computed d_{out}, commensurate with the noise characteristics. To compensate, L_{thresh} must be raised, with its ultimate selection dependent on the specific performance requirements of the fault detection algorithm.

Fault detection algorithms are subject to two types of errors: a False Alarm or a Missed Fault. A False Alarm (FA) is an error which occurs when there is no fault present in the system but the fault detection algorithm mistakenly signals the presence of a fault. A Missed Fault (MF) is an error which occurs when there is a fault present in the system but it is not detected by the fault detection algorithm. In our case, as is true with almost all fault detection algorithms, the probability of either of these errors is directly related to the value of L_{thresh}. We would ideally like to simultaneously minimize both the probability of a MF, (p(MF)), and the probability of a FA, (p(FA)). However, such a minimization is impossible because as the value of L_{thresh} varies, there exists an inherent tradeoff between p(MF) and p(FA).

No standard procedures exist to compare the relative importance of MF's versus FA's. In fact, their relative importance must be individually assessed for each application. One can easily envision different scenarios where MF's range from being critically important to operationally acceptable. Indeed, unless a MF were truly critical we certainly would not want to set L_{thresh} so low that we are continually shutting down the system in response to a slew of FA errors.

Regardless of the choice of L_{thresh} our fault detection algorithm will be unable to discern the difference between f and n. Thus, to completely eliminate the possibility of FA's our fault detection algorithm requires setting $L_{thresh} = 2\|n\|_{2\sigma}\|d_{in}\|_{2\sigma} + \|n\|_{2\sigma}^2$. This value is obtained by computing the maximum energy difference achievable between d_{out} and d_{in} given a noise input of energy $\|n\|_{2\sigma}^2$.

Conversely, completely eliminating the possibility of MF's requires setting L_{thresh} to a negative number. From Equation (14) such a value is intuitively unsettling because it *requires* that d_{out} have a prespecified amount of energy *less* than d_{in} in order for a fault not to be declared.

In reality the actual value of L_{thresh} will lie between these two extremes. For an arbitrary value of L_{thresh}, we would like to compute the minimum size of a fault f to assure detection by our algorithm. After some algebraic computations we find that all faults f such that

$$\|f\|_{2\sigma} \geq \|n\|_{2\sigma} + \|d_{out}\|_{2\sigma} + \sqrt{L_{thresh} + \|d_{in}\|_{2\sigma}^2}. \quad (16)$$

are detectable by our fault detection algorithm. From this equation it is clear that there are three key factors influencing the detectability of various faults: 1) the energy levels of the signals d_{in} and d_{out}, 2) the energy levels of f and n and 3) the value of the threshold L_{thresh}.

We can interpret the results of Equation (16) through a simple example. Suppose we have a case where both $\|d_{in}\|$ and $\|d_{out}\|$ are zero in steady state. Also, suppose the two signals f and n have the same energy levels and L_{thresh} is any nonzero number. Then it is conceivable (although unlikely) that n could completely cancel the signal f and it would appear as if no fault is present. In fact, to mask the presence of an existing fault, n need not completely cancel f, but it must only make it appear as if the combined signal $(n + f)$ has less energy than L_{thresh}. Therefore, to guarantee that a failure can be detected, the associated failure signal f must satisfy

$$\|f\|_{2\sigma} \geq \|n\|_{2\sigma} + \sqrt{L_{thresh}}. \quad (17)$$

From this equation it is clear that as either the energy level of n rises or L_{thresh} rises the energy requirements of f to assure detection increases. This does not mean that failure signals with less energy will not be detected. However, it does mean that the probability of detecting a fault associated with a fixed energy failure signal decreases as either L_{thresh} or the energy in n increases.

AN EXAMPLE

We now demonstrate the operation of our fault detection algorithm through an illustrative example. This example is based on the open loop system model shown in Figure 3. The plant model is represented by a family of plants as in Equation (??) with a nominal plant model given by $P_o(s) = \frac{1}{s+1}$ and uncertainty bound $\delta_{max} = 1$. The uncertainty block Δ consists of three high frequency second order flex modes and is given by

$$\Delta = \frac{200}{s^2 + 15s + 400} + \frac{200}{s^2 + 10s + 1600} + \frac{200}{s^2 + 10s + 4900}. \quad (18)$$

A frequency magnitude response plot of Δ is shown in Figure 5. Note that the magnitude response of Δ is below unity for all frequencies so $\|\Delta\|$ is indeed less than or equal to $\delta_{max} = 1$. The modeling uncertainty weighting function is given by W(s) = $\frac{s+.001}{s+5}$. This function was chosen to emphasize the modeling errors at high frequencies while

minimizing the errors at lower frequencies. The plant output measurement y is corrupted by an unknown noise n, and for our example we assume that $\|n\|_2^2 \leq .0025$.

For our fault detection algorithm we set $L_{thresh} = .5\|n\|_{2\sigma}\|d_{in}\|_{2\sigma} + \|n\|_{2\sigma}^2$. This is an intermediate value of L_{thresh} which does not preclude the possibility of MF's, but is also not extremely sensitive to FA's. Note also that this is a time-varying threshold which is dependent upon the computed values of d_{in}. We ran a variety of time domain simulations using a reference step input r of magnitude 1.1. We used the data for r and y to construct d_{in} and d_{out}. Finally, the weighted partial $\mathcal{L}_2$-norms (energies), E_{in} and E_{out}, of each of these two signals were computed using Equations (??) and (??).

In the first simulation we ran the system with no faults present and obtained the time histories shown in Figure 6. d_{in} (solid line) and d_{out} (dotted line) are shown in Figure 6a while E_{in} (solid line) and E_{out} (dotted line) are shown in Figure 6b. Note that as d_{in} reaches its steady state value about zero, the initial bump in E_{in} also decays to zero. This is explicitly due to the exponential forgetting factor built into $\bar{V}(t)$. Figure 6c displays the weighted partial energy of n. In a real system this value would not be available, but in our simulation we measured this value to ensure that our noise inputs adhered to our assumptions. Finally, the difference $E_{out} - E_{in}$ (solid line) and the threshold L_{thresh} (dotted line) are shown in Figure 6d. In this particular case the energy difference is always below L_{thresh} so the fault detection algorithm correctly concludes that no fault is present.

In the second simulation we configured the system such that a "large" failure corresponding to a step input of .11 would occur at time t = 2 seconds. The corresponding results, presented in an analagous manner to Figure 6, are shown in Figure 7. Note that the presence of the failure signal f is clearly evident in d_{out} and E_{out} for times greater than 2 seconds. In this case the energy difference shown in Figure 7d exceeds L_{thresh} almost immediately after the fault has occured (at roughly t = 2.35 seconds) so the fault detection algorithm very quickly and correctly concludes that a fault is present.

In the third time domain simulation we configured the system such that a "small" failure corresponding to a step input of .03 would occur at time t = 2 seconds. The corresponding results are shown in Figure 8. In this case the deviations of d_{out} and E_{out} from the no fault present case (Figure 6) are not as pronounced as in the "large" failure case (Figure 7). From the plots shown in Figure 8d we can see that it takes roughly three seconds after the fault occurs before the energy difference exceeds L_{thresh}. Thus, the fault detection algorithm correctly identifies the presence of this smaller fault but not as quickly as it identified the larger fault.

CONCLUSIONS AND RECOMMENDATIONS

In this paper we have described a new time-domain based algorithm for detecting faults in systems in the presence of modeling uncertainty. Our algorithm relies on processing true time domain system measurement data to analytically reconstruct key signals which characterize the modelling uncertainty. These signals are compared with apriori assumptions about the plant model to determine whether or not a fault is present. The algorithm is easy to implement on either an open or closed loop system and its use was demonstrated via a simple example. Several critical implementation issues were also addressed. An efficient numerical implementation of the algorithm was presented which featured reduced memory and computational requirements. In addition, a weighted partial energy criteria was employed which ensures that the algorithm emphasizes the newest data.

Certain crucial issues remain to be addressed. These include 1) optimal threshold selection strategies and the resulting tradeoff between False Alarm and Missed Fault error probabilities and 2) detecting and discerning a single failure from among a set of possible failures. These issues are currently under investigation.

REFERENCES

1. Emami-Naeini, A., M. Akhter, and S. Rock, "Effect of Model Uncertainty on Failure Detection: The Threshold Selector," IEEE Transactions on Automatic Control, Vol. 33, No. 12, pp. 1106-1115, 1988.

2. Friedland, B., "Maximum Likelihood Failure Detection of Aircraft Flight Control Sensors," Journal of Guidance, Control and Dynamics, Sept.-Oct. 1982, pp. 498-503.

3. Isermann, R., "Process Fault Detection Based on Modeling and Estimation Methods–A Survey", Automatica, Vol. 20, No. 4, pp. 387-404, 1984.

4. Kerr, T., "Decentralized Filtering and Redundancy Management for Multisensor Navigation", IEEE Transactions on Aerospace and Electronic Systems, Vol. 23, No. 1, pp. 83-119, 1987.

5. Krause, J. and P. Khargonekar, "Parameter Identification in the Presence of Non-Parametric Dynamic Uncertainty", Automatica, Vol. 26, pp. 113-124, Jan. 1990.

6. Patton, R., P. Frank and R. Clark (Eds.), "Fault Diagnosis in Dynamic Systems, Theory and Application", Prentice Hall, 1989.

7. Poolla, K., P. Khargonekar, A. Tikku, J. Krause, and K. Nagpal, "A Time- Domain Appraoch to Model Validation", to appear in IEEE Transactions on Automatic Control.

8. Smith, R. and J. Doyle, "Model Validation: A Connection Between Robust Control and Identification", IEEE Transactions on Automatic Control, Vol. 37, No. 7, pp. 942-952, 1992.

9. Smith, R., "Model Validation and Identification for Systems in H_∞ and l_1", Proceedings of the American Control Conference, pp. 2852-2856, June 1992.

10. Willsky, A., "A Survey of Design Methods for Failure Detection in Dynamic Systems", Automatica, Vol. 1976, pp. 601-611.

To: Pat Overstreet, Honeywell SRC, (612)951-7041
cc: Harry Kirschke, Honeywell SASSO, (813)539-5384
Charlie Poe, Honeywell SASSO, (408)756-2781
Bill Fouts, Honeywell SRC, (612)951-7034
John Weyrauch, Honeywell SRC, (612)951-7280
Jim Krause, Honeywell SRC, (612)951-7292
Lew Crouder, Lockheed Missiles and Space Co., (408)756-2781
Bill Edwards, Lockheed Missiles and Space Co., (408)756-2781

From: Mike Elgersma, Honeywell SRC, (612)951-7208
Date: June 01, 1993
Subject: Final Report to Honeywell SASSO for Lockheed THAADS work

Contract Name: THAAD/IAP Support
Purchase Order: 23070298
Contract/Project Number: F1183

Enclosed is the final report for work done by Honeywell SRC on THAAD.

Michael R. Elgersma

Date: June 01, 1993

Final Report for THAAD/IAP Support Executive Summary

This final report covers work done on May 18-21, 1993 to provide an independent analysis of a flexible missile control problem. During this time, I went to Lockheed to become familiar with an autopilot stability issue caused by flexibility of a thin missile. I evaluated the problem and concluded that the tail mounted gyro reduced the response of the first flex mode by a factor of ten. The extra gyro reduces a very significant flex problem to a marginal problem. The remainder of the problem can be handled by some combination of flex filtering, passive damping, and employing aero devices to move the c.p. back during the high dynamic pressure portion of flight.

The following section gives several formulas for the unstable aero mode, the size and location of the first flex mode, as well as an envelope for the remaining flex modes. These formulas are useful for determining the fundamental physical tradeoffs in designing a uniform cylindrically shaped missile. I would like to thank Bill Edwards, Robert Felder, Conrad Woo, John Sesak, and Doug Discher for there help in providing me the necessary information for this project.

Date: June 01, 1993

Final report for THAAD/IAP Technical section:

Summary

The most important parameters in the control of an unstable flexible missile are:

$\frac{\omega_{aero}}{\omega_{flex_1}}$ = the ratio of the unstable aero frequency and the first flex mode frequency,

$\frac{\omega_{aero}}{\omega_{actuator}}$ = the ratio of the unstable aero frequency and the actuator bandwidth,

$\frac{\Delta\Psi}{\Psi}$ = uncertainty in the flex mode shapes,

ζ = the flex damping ratio,

ζ_c = the controller damping ratio,

and the size of the resulting flex resonance peaks in the Bode plots.

With only one gyro mounted at the nose:

$$\text{Bode Plot peak} = \frac{\sqrt{2}\omega_{aero}}{\omega_{flex_1}}\frac{\zeta_c}{\zeta} = \frac{1}{\nu_1}\sqrt{24\left[\frac{\bar{q}}{E}\right]\left[\frac{L^2}{Rt}\right]\left[\frac{c.p.-c.g.}{L}\right]C_{N_\alpha}}\,\frac{\zeta_c}{\zeta} \qquad (1)$$

With blended fore and aft gyros,

$$\text{Bode Plot peak} = \frac{\sqrt{2}\omega_{aero}}{\omega_{flex_1}}\,\frac{\zeta_c}{\zeta}\,\left|1\tfrac{1}{2}\frac{\Delta\Psi}{\Psi}\right| = \frac{1}{\nu_1}\sqrt{24\left[\frac{\bar{q}}{E}\right]\left[\frac{L^2}{Rt}\right]\left[\frac{c.p.-c.g.}{L}\right]C_{N_\alpha}}\,\frac{\zeta_c}{\zeta}\,\left|1\tfrac{1}{2}\frac{\Delta\Psi}{\Psi}\right| \qquad (2)$$

where

L = the length of the missile

R = the radius of the missile

t = the skin thickness of the missile

ρ_m = the density of the entire missile

c.p. = the center of pressure of the missile

c.g. = the center of gravity of the missile

C_{N_α} = the aerodynamic normal coefficient ≈ 2

ν_1 = flex boundary condition coefficient ≈ 22

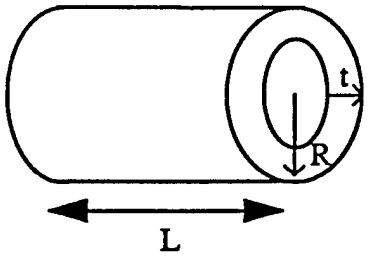
E = Young's Modulus of the missile skin

$\bar{q}$ = dynamic pressure

ζ the flex damping ratio (between .01 and .05)

ζ_c = the controller damping ratio (between .2 and 1)

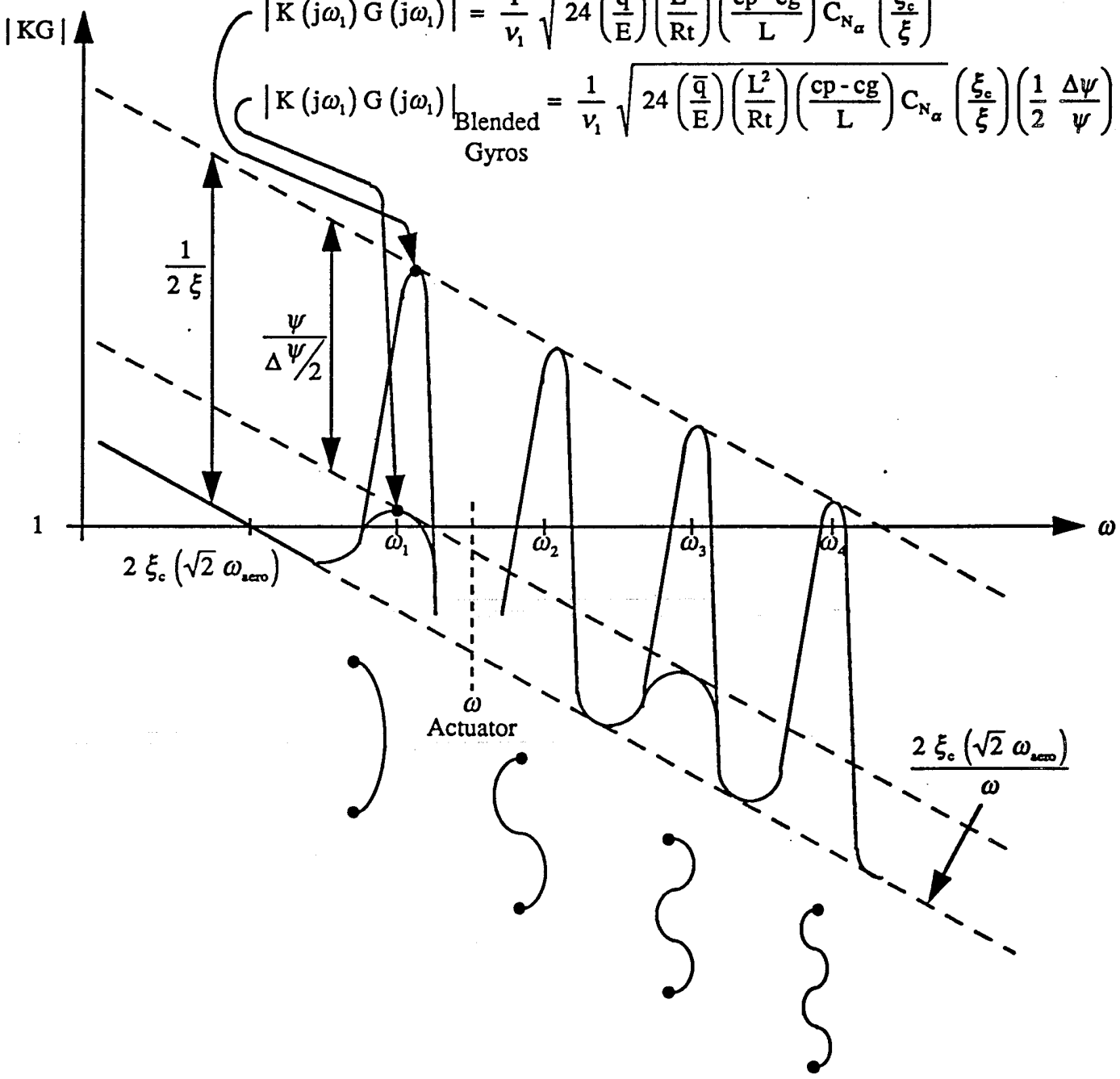
Bode Plot of Long Thin Missile System Stabilized by a PD Controller



$t \ll R \ll L$

$$|K(j\omega_1)\,G(j\omega_1)| = \frac{1}{\nu_1}\sqrt{24\left(\frac{\bar{q}}{E}\right)\left(\frac{L^2}{Rt}\right)\left(\frac{cp-cg}{L}\right)C_{N_\alpha}\left(\frac{\xi_c}{\xi}\right)}$$

$$|K(j\omega_1)\,G(j\omega_1)|_{\text{Blended Gyros}} = \frac{1}{\nu_1}\sqrt{24\left(\frac{\bar{q}}{E}\right)\left(\frac{L^2}{Rt}\right)\left(\frac{cp-cg}{L}\right)C_{N_\alpha}\left(\frac{\xi_c}{\xi}\right)\left(\frac{1}{2}\,\frac{\Delta\psi}{\psi}\right)}$$



Flex Dynamics

The geometry of the THAAD missile is closely approximated as a free-free uniform thin annulus of length L, radius R, skin thickness t (with $t << R << L$), density ρ_m, and Young's modulus E. Only the metal skin (of thickness $t << R$) contributes to the stiffness, while the entire cylinder (of thickness R) contributes to the density. As the fuel burns (uniformly along the length of the missile), ρ_m decreases.

The uniformity of the missile allows the use of a simple analytic formula to approximate the frequency of the first few flex modes, ω_{flex_n}. The following formula is taken from W. C. Young, "Roark's Stress and Strain," Sixth Edition, McGraw-Hill, 1989.

$$\omega_{flex_n} = \nu_n \sqrt{\left[\frac{E\pi R^3 t}{(\rho_m L \pi R^2) L^3}\right]} = \nu_n \sqrt{\frac{ERt}{\rho_m L^4}} \tag{3}$$

where

$\nu_1 \approx 22.4$, $\nu_2 \approx 61.7$, $\nu_3 \approx 121$, $\nu_4 \approx 200$, and $\nu_5 \approx 299$.

The damping ratio, ζ, is approximately .01 since there are almost no joints in the structure.

The odd mode shapes have the same displacement but opposite slope at the nose and tail. The even mode shapes have opposite displacement but same slope at the nose and tail.

Let τ_c be the control torque applied at the tail (x=0) of the missile using thrust vector control ($\tau_c \approx \frac{L}{2}$ Thrust $\sin(\delta)$). Let $\Psi_i(x)$ be the mode slope of mode i at the point x (x=0 at tail, x=L at nose). The mode slope gives the coupling between torque inputs and the flex response, as well as from the flex response to the angular rate measurements. The flexible body transfer function from control torque to pitch rate (at the point x) can be approximated as:

$$\frac{\dot{\theta}(x)}{\tau_c} = \left(\frac{1}{J}\right)\left[\sum_i \frac{\Psi_i(x)\, s\, \Psi_i(0)}{s^2 + 2\zeta\omega_{flex_i} s + \omega_{flex_i}^2}\right] \tag{4}$$

Since the missile is a uniform shape, all the mass participates in each mode, therefore $\Psi_i(x)$ is about size 1, i.e.

$$\Psi_i(0) \approx 1.0$$
$$\Psi_i(L) \approx (-1)^i \left(1 + \frac{\Delta\Psi}{\Psi}\right) \tag{5}$$

where $\Delta\Psi$ represents the uncertainty in the match between the mode slope at the fore and aft locations. Since the missile is nearly uniform in shape, the slopes should match to within 10%, i.e. $\left|\frac{\Delta\Psi}{\Psi}\right| < .1$ for this missile.

Rigid Body Dynamics due to Aero

The aero torque due to flying through air of density ρ at speed V is approximately given by:

$$\tau_{aero} = (\tfrac{1}{2}\rho V^2)(\pi R^2)(\text{c.p.} - \text{c.g.})\, C_{N_\alpha}\, (\Theta - \gamma) \tag{6}$$

We will assume that the flight path angle γ changes much slower than the vehicle pitch angle Θ, and will linearize about $\theta = \Theta - \gamma$. Then the rigid body pitch dynamics can be approximated with the following second order system:

$$J\,\ddot{\theta} = (\tfrac{1}{2}\rho V^2)(\pi R^2)(\text{c.p.} - \text{c.g.})\, C_{N_\alpha}\, \theta \tag{7}$$

If we rewrite this as:

$$\ddot{\theta} = (\omega_{aero})^2\, \theta \tag{8}$$

then the unstable (c.p. > c.g.) aero frequency is approximately given by:

$$\omega_{aero} = \sqrt{\left[\frac{(\tfrac{1}{2}\rho V^2)(\pi R^2)(\text{c.p.} - \text{c.g.})\, C_{N_\alpha}}{J}\right]} \tag{9}$$

The moment of inertia of a uniform thin (R << L) cylinder of length L, radius R, and density ρ_m is approximately given by:

$$J = \frac{\rho_m\, \pi R^2\, L^3}{12} \tag{10}$$

which gives

$$\omega_{aero} = \sqrt{\frac{12(\frac{1}{2}\rho V^2)(c.p. - c.g.)\ C_{N_\alpha}}{\rho_m L^3}}$$
$$= \frac{V}{L}\sqrt{6\frac{\rho}{\rho_m}\frac{c.p. - c.g.}{L}\ C_{N_\alpha}} \quad (11)$$

The rigid body transfer function from control torque to angular rate is:

$$\frac{\dot{\theta}}{\tau_c} = (\frac{1}{J})\left[\frac{s}{s^2 - \omega_{aero}^2}\right] \quad (12)$$

Combined Flex and Rigid Dynamics

The ratio of equations 11 and 3 gives:

$$\frac{\omega_{aero}}{\omega_{flex_n}} = \frac{1}{\nu_n}\sqrt{12\left[\frac{\bar{q}}{E}\right]\left[\frac{L^2}{Rt}\right]\left[\frac{c.p. - c.g.}{L}\right] C_{N_\alpha}} \quad (13)$$

The combined rigid and flexible body transfer function from control torque to pitch rate (at the point x) can be approximated as:

$$\frac{\dot{\theta}(x)}{\tau_c} = (\frac{1}{J})\left[\frac{s}{s^2 - \omega_{aero}^2} + \sum_i \frac{\Psi_i(x)\ s\ \Psi_i(0)}{s^2 + 2\zeta\omega_{flex_i}s + \omega_{flex_i}^2}\right] \quad (14)$$

Actuators

The THAAD actuator frequency is currently between the first and second flex modes.

$$\omega_{flex_1} < \omega_{actuator} < \omega_{flex_2} \quad (15)$$

The transfer function given in equation 14 must be modified at frequencies above the actuator frequency.

Controller

Assume there are rate gyros at locations x_{rg_j} and attitude gyros at locations x_{ag_k}. In order to

robustly control the unstable aero rigid body mode, the controller must have approximately twice as much torque as the unstable aero torque.

$$\omega_c^2 = 2\omega_{aero}^2 \tag{16}$$

A simple PD controller (with damping ratio $.2 < \zeta_c < 1$) would then have the form

$$\tau_c = -J \left[2\zeta_c\omega_c \sum_j [c_j \dot{\theta}(x_{rg_j})] + \omega_c^2 \sum_k [d_k \theta(x_{ag_k})] \right] \tag{17}$$

where the gyros have been blended to give the correct rigid body information:

$$\sum_j c_j = 1$$

$$\sum_k d_k = 1$$

At high frequency (ie $\omega \gg \omega_c$) the rate term dominates the position term, so at high frequency:

$$\tau_c \approx -J\ (2\zeta_c\omega_c) \sum_j [\dot{\theta}(x_{rg_j})] \tag{18}$$

combining equations 14 and 18 gives the loop gain of the combined controller, K(s), and plant, G(s):

$$K(s)G(s) = \left[\frac{2\zeta_c\omega_c\, s}{s^2 - \omega_{aero}^2} + \sum_i \sum_j \frac{c_j \Psi_i(x_{rg_j})\, 2\zeta_c\omega_c\, s\, \Psi_i(0)}{s^2 + 2\zeta\omega_{flex_i} s + \omega_{flex_i}^2} \right] \tag{19}$$

Fore and Aft Rate Gyros

If only one gyro is used, it must be placed in the nose, since the rest of the vehicle separates after burnout. For controlling the flexible booster, it would be better to have the rate gyro mounted at the same location as the actuator (at the tail) since then the measurement and the applied torque would be collocated and would remain in phase for all flex modes. This allows the controller to remove energy by making the torque proportional to the sensed angular rate. This would ensure that all flex modes were phase stable, except for the fact that the actuator dynamics ruin this argument above the actuator frequency. Consequently, it is safer to gain stabilize the flex modes.

By blending the signals from a nose and tail gyro, the odd flex modes will nearly cancel, since they have opposite mode slopes at the fore and aft locations. This not only stabilizes the odd flex modes, but also reduces the ringing that would be present if we relied only on a single collocated gryo.

If one rate gyro is placed at the nose, ($x_{rg_1} = L$), and the second rate gyro is placed at the tail, ($x_{rg_2} = 0$), then

$$\Psi_i(x_{rg_1})\ \Psi_i(0) = \Psi_i(L)\ \Psi_i(0) = (-1)^i\ (1 + \frac{\Delta\Psi}{\Psi})$$

and (20)

$$\Psi_i(x_{rg_2})\ \Psi_i(0) = \Psi_i(0)\ \Psi_i(0) = 1$$

Since the missile is very uniform, the mode slopes are known to within 10%,

$$|\frac{\Delta\Psi_1}{\Psi_1}| < .1 \tag{21}$$

The loop gain with a fore and aft gyro is given by:

$$-K(s)G(s) = \left[\frac{2\zeta_c\omega_c\, s}{s^2 - \omega_{aero}{}^2} + \sum_i \frac{((-1)^i(1 + \frac{\Delta\Psi}{\Psi})c_1 + c_2)\ 2\zeta_c\omega_c\, s}{s^2 + 2\zeta\omega_{flex_i}s + \omega_{flex_i}{}^2}\right] \tag{22}$$

with $c_1 + c_2 = 1$.

Evaluating the above expression at the flex frequencies gives the size of the peaks:

$$\begin{aligned} -K(j\omega_{flex_n})G(j\omega_{flex_n}) &\approx \left[(-1)^n c_1(1 + \frac{\Delta\Psi}{\Psi}) + c_2\right] \frac{\zeta_c\omega_c}{\zeta\omega_{flex_n}} \\ &\approx \left[(-1)^n\, c_1(1 + \frac{\Delta\Psi}{\Psi}) + c_2\right] \frac{\zeta_c\sqrt{2}\omega_{aero}}{\zeta\omega_{flex_n}} \end{aligned} \tag{23}$$

If $c_1 = 0$ and $c_2 = 1$, then the size of the flex peaks are:

$$|K(j\omega_{flex_n})G(j\omega_{flex_n})| \approx \frac{\zeta_c\sqrt{2}\omega_{aero}}{\zeta\omega_{flex_n}} \tag{24}$$

If $c_1 = .5 = c_2$, then the size of the odd flex peaks are:

$$|K(j\omega_{flex_n})G(j\omega_{flex_n})| \approx |½ \frac{\Delta\Psi}{\Psi}| \frac{\zeta_c\sqrt{2}\omega_{aero}}{\zeta\omega_{flex_n}} \quad (25)$$

Combining equation 13 with equations 24 and 25 gives equations 1 and 2 in the summary.

Recall that we are assuming that the controller rate term dominates the controller attitude term by a factor of $\frac{\omega_{flex_n}}{\omega_{aero}}$. The blending is done only on the rate gyros, so if the loop gain in equation 25 drops below the attitude loop gain. i.e. $|\frac{\Delta\Psi}{\Psi}| < \frac{\omega_{aero}}{\omega_{flex_n}}$, then the flex signals coming through the unblended attitude loop may become larger than the rate loop.

Detuning the Blended Rate Gyros

Depending on whether $\Delta\Psi$ is positive or negative, setting $c_1 = .5 = c_2$ could result in either the fore or aft gyro dominating. In order to design a flex filter to further reduce the first flex peak, it is necessary to know the phase of the signal, therefore it is necessary to know which gyro is dominating. It may be necessary to set $c_1 = .45$ and $c_2 = .55$ to ensure that the aft gyro dominates. This will slightly increase the size of the first flex peak, but will ensure proper phasing of the blended gyro signals when there is 10% uncertainty in the mode slopes.

Flex Filter Design Suggestions

The flex filters should add as little phase lag as possible. If the gain is to be rolled off uniformly above some frequency, then a Butterworth filter should be used to minimize phase loss. If the flex frequencies (which increase as fuel is burned) are known well enough, then the flex modes that are not phase stable can be notched out. A narrow notch filter introduces less phase lag than a broadband filter.

Analyzing Actuator Nonlinearities

The actuator nonlinearity (due to the backlash, hysteresis, etc. of the EMA gearing) can be analyzed using describing function analysis. If the gear-drive has enough friction that it is not backdrivable, then the aero loads on the actuator can be neglected.